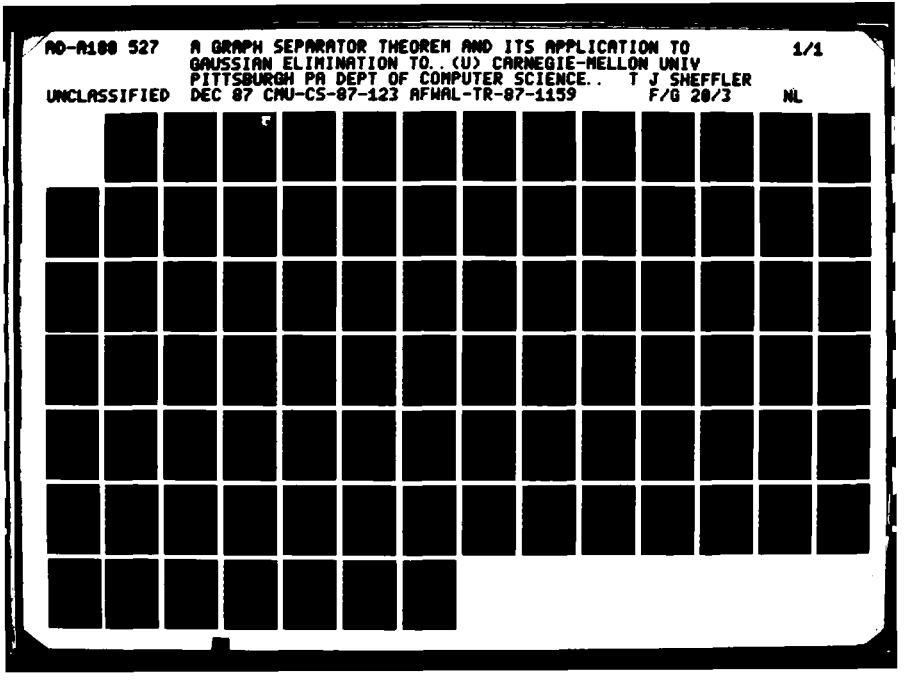
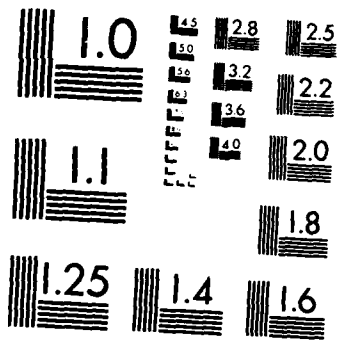

AD-A188 527 A GRAPH SEPARATOR THEOREM AND ITS APPLICATION TO GAUSSIAN ELIMINATION TO.. (U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.. T J SHEFFLER
1/1
UNCLASSIFIED DEC 87 CMU-CS-87-123 AFWAL-TR-87-1159 F/G 20/3 NL



PHOTOGRAPH THIS SHEET

AD-A188 527

DTIC ACCESSION NUMBER

LEVEL

1

INVENTORY

AFWAL-TR-87-1159

DOCUMENT IDENTIFICATION

Dec 1987

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS GRA&I	X
DTIC TAB	☐
UNANNOUNCED	☐
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A-1	

DISTRIBUTION STAMP

QUALITY INSPECTED 2

DTIC ELECTE FEB 08 1988 D E

DATE ACCESSIONED

DATE RETURNED

88 2 05 108

DATE RECEIVED IN DTIC

REGISTERED OR CERTIFIED NO.

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

DTIC FORM 70A DEC 83

DOCUMENT PROCESSING SHEET

PREVIOUS EDITION MAY BE USED UNTIL STOCK IS EXHAUSTED.

AFWAL-TR-87-1159

A GRAPH SEPARATOR THEOREM AND ITS APPLICATION TO GAUSSIAN ELIMINATION TO OPTIMIZE BOOLEAN EXPRESSIONS FOR PARALLEL EVALUATION

Thomas J. Sheffler

Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213-3890

December 1987

Interim

AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Chahira M. Hopper

CHAHIRA M. HOPPER
Project Engineer

Richard C Jones

RICHARD C. JONES
Ch, Advanced Systems Research Gp
Information Processing Technology Br

FOR THE COMMANDER

Edward L. Gliatti

EDWARD L. GLIATTI
Ch, Information Processing Technology Br
Systems Avionics Div

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT , Wright-Patterson AFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU-CS-87-123	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFWAL-TR-87-1159
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Air Force Wright Aeronautical Laboratories AFWAL/AAAT-3
6c. ADDRESS (City, State, and ZIP Code) Computer Science Dept Pittsburgh PA 15213-3890	7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6543
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33615-84-K-1520
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
61101E	4976	00	01

11. TITLE (Include Security Classification)
A Graph Separator Theorem and Its Application to Gaussian Elimination to Optimize Boolean Expressions for Parallel Evaluation

12. PERSONAL AUTHOR(S)
Thomas J. Sheffler

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM _____ TO _____	1987 December	87

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Gaussian elimination, which has been shown to be applicable to the solution of problems in many different domains, is the technique used by COSMOS to symbolically analyze digital MOS networks for their behavior in terms of Boolean expressions. While pivot selection algorithms are known which minimize the total number of operations required to solve a system, this report will focus on pivot selection algorithms that result in expressions of small *depth*, from which fine-grained parallelism may be extracted.

A graph theoretic approach to Gaussian elimination is adopted which allows

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Chahira M. Hopper	22b. TELEPHONE (Include Area Code) (513) 255-7865
	22c. OFFICE SYMBOL AFWAL/AAAT-3

DD Form 1473, JUN 86 *Previous editions are obsolete*

Block 19 (Continued)

the structure of sparse systems to be clearly examined, and an elimination ordering based on graph separators is shown to result in expressions of small depth. This report proposes an algorithm related to Gaussian elimination which characterizes graphs in terms of *decomposition rules* and shows that for graphs which may be reduced by an elimination ordering that results in low total complexity, a reordered elimination sequence may result in expressions of small depth.

Acknowledgements

My sincere appreciation goes to my friends and family who have been supportive of me in these last months and have put up with my periods of irritability when things were not going right. A special thanks is extended to D. Sleator whose assistance with the proof of Chapter 5 was most appreciated, and to R. Heuler whose careful editing was an enormous help. I am indebted to my advisor, R. Bryant, who suggested this topic and guided me in my research. While being patient enough to allow me to explore areas which would eventually lead nowhere, he was always there to help me get back on track. I am grateful to him for his guidance and enthusiasm throughout the course of this work.

Contents

1 Introduction 1

1.1 COSMOS 3

1.2 Gaussian Elimination 5

1.3 Expression Representation and Pivot Selection 6

1.4 Parallel Architectures / Simulation Engines 7

2 Gaussian Elimination 11

2.1 Introduction 11

2.2 A Graph Theoretic Approach 13

2.3 Path Problems on Graphs 15

2.4 Choosing an Elimination Ordering 18

3 Generalized Nested Dissection 22

3.1 Graph Separators 22

3.2 Elimination Ordering Algorithms 24

3.2.1 Nested Dissection 24

3.2.2 Automatic Nested Dissection 26

3.2.3 Generalized Nested Dissection - I 26
3.2.4 Generalized Nested Dissection - II 28
3.3 Separator Trees 28
3.4 Complexity Versus Structure 30

4 Separators for Series-Parallel Graphs **34**
4.1 Series-Parallel Reduction Rules 34
4.2 Separators of SP Graphs 37
4.3 An Elimination Ordering for SP Graphs 40

5 Separators for Arbitrary Graphs **46**
5.1 Elimination Cliques 46
5.2 Reduction Rules for Arbitrary Graphs 49
5.3 A Separator Theorem for a Bipartite Tree 51
5.4 Using a Decomposition Tree to find Separators 54
5.4.1 Bounding the Number of Children 56
5.5 A Nested Dissection Ordering Algorithm 57
5.6 A Bound on Complexity 60
5.7 A Bound on Depth 60
5.8 SP and GSP Graphs Revisited 63
5.9 Experimental Results 64

6 Discussion **68**
6.1 Future Considerations 69

A Graph Theoretical Definitions **70**

B Various Channel Graphs **72**

Bibliography **75**

List of Figures

1 COSMOS Implementation 3
2 A Simple Graph of Seven Vertices 7
3 DAG Resulting from Left to Right Elimination Ordering 8
4 DAG Resulting from Nested Dissection Ordering 9

5 Solution of a System of Equations Under Two Permutations 12
6 The Fill-in Resulting From Two Different Elimination Orders 14
7 Solution of a Boolean System by Gaussian Elimination 17
8 The General Series-Parallel Production Rules 19
9 Examples of GSP Graphs 19
10 Non-GSP Graphs 20

11 A 1-Separator for a Tree 23
12 A Grid Graph of Size $n = k \times k$ With Separator Set Indicated 25
13 A Graph and Its Separator Tree 29
14 A Graph Resembling a Binary Tree 31
15 The Resulting DAG With an Ordering Due to the Minimum Degree Algorithm 32
16 The Resulting DAG With an Ordering Due to Nested Dissection . . . 33

17 Series and Parallel Reduction Rules 35
18 Construction of a Decomposition Tree for a SP graph 36
19 Edge Labeling for the Reduction of SP Graphs 37
20 Labeling for Each Node in a Decomposition Tree 38
21 A Decomposition Tree After Labeling 38

LIST OF FIGURES

22 Recursive Algorithm for Finding 2-Separator of a SP Graph 39
23 Nested Dissection Elimination Ordering Algorithm for SP Graphs . 42
24 A Sample Graph 43
25 The DAG Resulting from the Generalized Nested Dissection Algorithm 44
26 The DAG Resulting from the Minimum Degree Algorithm 45

27 A Single Reduction Rule for SP graphs 48
28 Reduction Rules for Arbitrary Graphs 49
29 A Decomposition Tree Fragment 49
30 Elimination of v_j and the Resulting Decomposition Tree Fragments . 50
31 The Resulting Decomposition Tree 50
32 An Algorithm to Produce a Decomposition Tree for an Arbitrary Graph 52
33 A Red-Black Tree 53
34 Algorithm to Find a Black Separator Node 55
35 A Reduction Rule of Order Two 56
36 Modified Algorithm to Produce Decomposition Tree for an Arbitrary Graph 58
37 Elimination Ordering Algorithm for Generalized Decomposition Trees 59
38 The Depth of DAGs for a Clique 61
39 A Separation Set and Two External Vertices 62

40 A Linear Chain Graph of Ten Vertices 73
41 A Sixteen-Bit Shifter 73
42 A Sixteen-Bit Logical Shifter 73
43 Seradd.a 74
44 Seradd.b 74
45 Channel Graph for a Sixteen-Bit RAM Cell 74

List of Tables

1 Comparison of Minimum Degree and Automatic Nested Dissection . 27

2 Results for Linear Chains of Varying Length 65
3 Results for 16-bit shifters 65
4 Results for 16-bit logical shifters 66
5 Results for RAMs of Varying Sizes 66
6 Results for Parity Ladders of Varying Number of Vertices 67
7 A Few Random Networks 67

Chapter 1

Introduction

A recent development in the continuing search for faster methods of circuit simulation has been the advent of special-purpose hardware dedicated to simulation. With a large number of small elements working in parallel, increases in speed of two to four orders of magnitude over general-purpose computers are not uncommon. While this equipment is well-suited to simulation at the gate level, it has not been very successful at simulating MOS networks due to the bidirectionality of MOS transistors. A variety of special-purpose hardware elements have been designed to capture this characteristic of MOS transistors with limited success. A more promising approach has been demonstrated by the COSMOS (Compiled Simulator for MOS) simulator, which symbolically analyzes the behavior of a MOS circuit and then represents it in purely Boolean terms. The expressions generated by COSMOS can then be evaluated on hardware designed for gate-level simulation.

The Boolean expressions generated are multi-level with shared subexpressions. While simplification techniques for less complex Boolean expressions exist, none are applicable to expressions of this type. When mapping these expressions onto special-purpose hardware, a new definition of *simplicity* must be made. Just as the limiting factor for the performace of logic is its critical path, the limiting factor for the performance of a MOS simulator using Boolean expressions on highly parallel hardware is the *depth* of the expressions. However, optimization of Boolean expressions to minimize depth is an issue that has not been widely discussed.

Gaussian elimination is a technique that has continually proved to be useful for problem solving in areas other than systems of linear equations. In the COSMOS software, Gaussian elimination is used in the analysis of subnetworks to generate Boolean expressions. In this process, pivot selection is a crucial issue and many methods have been proposed to minimize the total number of algebraic operations performed. The order in which pivots are selected also affects the *structure* of operations, an effect which has not been studied in depth.

Presented in this report is a pivot selection algorithm that guarantees that the expressions generated by Gaussian elimination will exhibit *balance* and will therefore have small depth. The remainder of this chapter is dedicated to explaining the motivation of this research in relation to the COSMOS project. Chapter 2 discusses Gaussian elimination and introduces a graph theoretic view of the algorithm. Pivot selection is shown to be important for keeping the total complexity of the solution of a sparse system low.

Graph separator theorems and nested dissection pivot selection algorithms are the subject of Chapter 3, where it is shown that an ordering based on nested dissection results in expressions of small depth. Chapter 4 analyzes series-parallel graphs and presents decomposition trees, which are shown to be useful for finding separators in such graphs. With these separators, it is easy to produce an elimination ordering for a series-parallel graph that results in small depth.

In Chapter 5, an algorithm is described that allows the construction of decomposition trees for arbitrary graphs. These trees are constructed through a procedure very similar to Gaussian elimination. A heuristic strategy is used for this first pass, but it is shown that an elimination ordering that results in low total complexity can be used to find an elimination ordering that results in small depth of the expressions generated. The method has applicability to any system solved by Gaussian elimination but is especially suited to those systems that arise in the analysis of MOS networks. Finally, Chapter 6 briefly discusses the outcome of this work and suggests areas for future research. Also, Appendix A is provided to explain some basic graph theoretical terms used in this report.

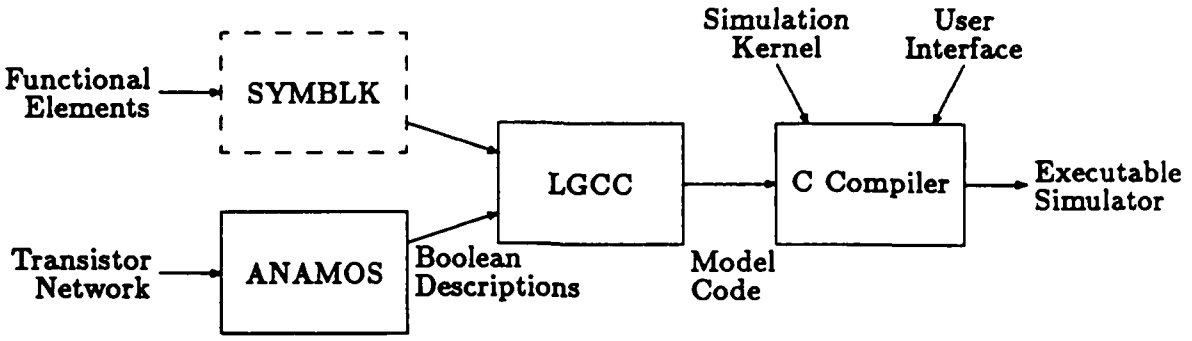


Figure 1: COSMOS Implementation

1.1 COSMOS

COSMOS consists of a set of programs configured as shown in Figure 1. The symbolic analyzer ANAMOS accepts as input the switch-level representation of a MOS circuit and partitions it into a set of channel-connected subnetworks. It then derives a Boolean representation of the behavior of each subnetwork. A second program, LGCC, translates this Boolean representation into a set of C language evaluation procedures plus declarations of data structures describing the interconnections of the networks. Finally, the code produced by LGCC, together with the simulation kernel and user interface code, are compiled to generate the simulation program. The resulting program appears to the user much like an ordinary simulator, except that the network is already loaded at the start of execution. The simulator implements a block-level, event-driven scheduler, in which blocks correspond to subnetworks. Processing an event involves calling the appropriate procedure to recompute the outputs of a block.

Unlike programs that operate directly on the transistor-level description during simulation, COSMOS preprocesses the transistor network to produce a Boolean description. This description, formulated by ANAMOS, captures all aspects of switch-level networks, including bidirectional transistors, stored charge, different signal strengths, and indeterminate (X) logic values.

The most novel aspects of COSMOS are found in ANAMOS. The transistor network is partitioned into channel-connected subnetworks and the steady-state response of each subnetwork is derived separately. Each subnetwork corresponds to a component of the undirected graph having as vertices the storage nodes and as edges the pairs of nodes connected by transistor sources and drains. This partitioning describes the static connections in the network; i.e. those independent of transistor state.

Typical MOS circuits partition into many small subnetworks, although some subnetworks can be quite large. Within a subnetwork, the behavior can be complex and difficult to analyze due to the bidirectionality of transistors and the variety of ways in which state is formed in a MOS circuit. The interactions between subnetworks, however, are more straightforward. Each subnetwork acts as a sequential logic element having as input the input nodes connected to transistor sources and drains, plus the gate nodes of the transistors. The subnetwork state is stored as charge on the storage nodes. Its outputs are those nodes that are gate nodes of transistors in other subnetworks.

To cast the switch-level model in terms of Boolean operations, a logic value $y \in \{0, 1, X\}$ is represented by a "dual rail" Boolean encoding, $y.1, y.0 \in \{0, 1\}$, as shown below:

y	$y.1$	$y.0$
1	1	0
0	0	1
X	1	1

With a Boolean encoding of the state values, the problem of symbolic analysis can be defined as follows. For each node n, introduce Boolean variables $n.1$ and $n.0$ to represent the possible encoded values of the node state. For each node n, ANAMOS derives Boolean formulas $N.1$ and $N.0$ in terms of the set of node state variables. These formulas represent the encoded value of the steady-state response at each node as a function of the initial node states.

Switch-level networks resemble classical contact networks in that both are composed of bidirectional switching elements. Shannon [28] first developed techniques for analyzing a contact network symbolically. In his method, each contact is labeled with a Boolean literal and the conditions under which a path may form between designated pairs of terminals are formulated as a Boolean expression. This idea serves as the conceptual basis of ANAMOS, although MOS circuits require a more complex method of analysis. Furthermore, most of the methods presented in the contact network literature are not particularly well suited to computerized applications for large circuits.

1.2 Gaussian Elimination

The contact network analysis problem can be formulated as the solution of a system of Boolean equations [7]. Solution of this system relates closely to the problem of finding expressions that describe all paths between vertices in a directed graph. Tarjan [29] has shown that a generalized form of Gaussian elimination can solve a large class of path problems, including contact network analysis. When solving a system of Boolean equations by Gaussian elimination, Boolean operations $\wedge$ and $\vee$ replace the conventional arithmetic operations.

Gaussian elimination proceeds in two parts: forward elimination and backsolving. Forward elimination propagates information about paths forward to a single vertex. Backsolving distributes information to the vertices previously eliminated. Forward elimination consists of repeatedly selecting a vertex v (the "pivot") for removal from the graph. The elimination of vertex v involves propagating its label to each neighbor u through their shared edge. Furthermore, the label of an edge spanning each pair of neighbors u and w may be updated as well. Backsolving proceeds by adding vertices back to the graph in the reverse order of their elimination. The value on each vertex is computed by summing the effects of all neighbors eliminated after v.

Gaussian elimination has a distinct advantage over iterative methods for symbolic analysis. Being a direct method, it requires no testing for convergence. Symbolic analysis can proceed by simply constructing Boolean formulas in terms of operations $\wedge$ and $\vee$ in accordance with the elimination steps. A direct method avoids the need to test formulas for equivalence, an NP-hard problem.

1.3 Expression Representation and Pivot Selection

ANAMOS represents a Boolean formula as a directed acyclic graph (DAG). A DAG resembles a parse tree whose leaves represent variables or constants and whose internal nodes represent binary Boolean operations. In a DAG, however, a given subgraph may be shared by several branches, yielding a more compact representation [4]. During the analysis of a subnetwork, ANAMOS constructs a single DAG with multiple roots, one root for each vertex and edge in the channel graph representing the subnetwork.

Upon completion of the execution of the Gaussian elimination algorithm, the steady-state response of the subnetwork has been computed. The DAG contains one root for each vertex of the channel graph, and the expression indicated by that root gives the steady state response at that vertex in the graph.

The total number of nodes in the DAG, as well as its structure, is determined by the order of operation on vertices in the Gaussian elimination procedure. Figure 2 shows a simple graph that could represent a pass transistor network of six transistors, or a network consisting of six switches. The vertices and edges are labeled with symbolic values. In the Gaussian elimination procedure, the vertices in the graph could be operated on in a number of different orders.

If the vertices are eliminated from left to right, then the DAG of Figure 3, with depth of 24, results. Alternatively, if the vertices are eliminated in an order that resembles that of a nested dissection algorithm, the DAG of Figure 4 results. This DAG has a depth of only 11 operations, less than half of the other one. Each

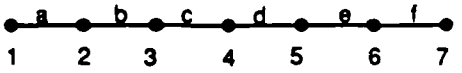


Figure 2: A Simple Graph of Seven Vertices

DAG indicates the same total number of operations, but their structure is different. The DAG of Figure 3 represents the solution of a system in which the operators must be applied to values sequentially, whereas the DAG of Figure 4 indicates a situation in which some parallelism could be exploited, given a machine capable of such an evaluation strategy. For longer chains of vertices, the results are more dramatic. The left-to-right elimination ordering will give expressions whose depth grows as $O(n)$, while a nested dissection algorithm will give expressions that grow as $O(\lg n)$.

1.4 Parallel Architectures / Simulation Engines

The COSMOS simulator as designed runs on a general-purpose computer. During simulation, the DAGs representing a subnetwork are evaluated for new values at each simulation clock cycle. An event-driven scheduling algorithm is used to determine those subnetworks that need to be evaluated, since some subnetworks will not require evaluation on every clock cycle. The total amount of time required to simulate a single clock cycle is a function of the total number of operations in each of the DAGs, and the number of subnetworks needing to be evaluated.

Special-purpose processors have been developed to accelerate tasks of logic gate simulation [9,1,6]. Although special-purpose processors for switch-level simulation have been designed and constructed [8,12], they require a substantial amount of specialized hardware. It is unlikely that they will ever achieve the cost/performance levels of processors that support only gate-level evaluation. These gate-level simulation processors use many simple function evaluators in parallel to achieve their speed.

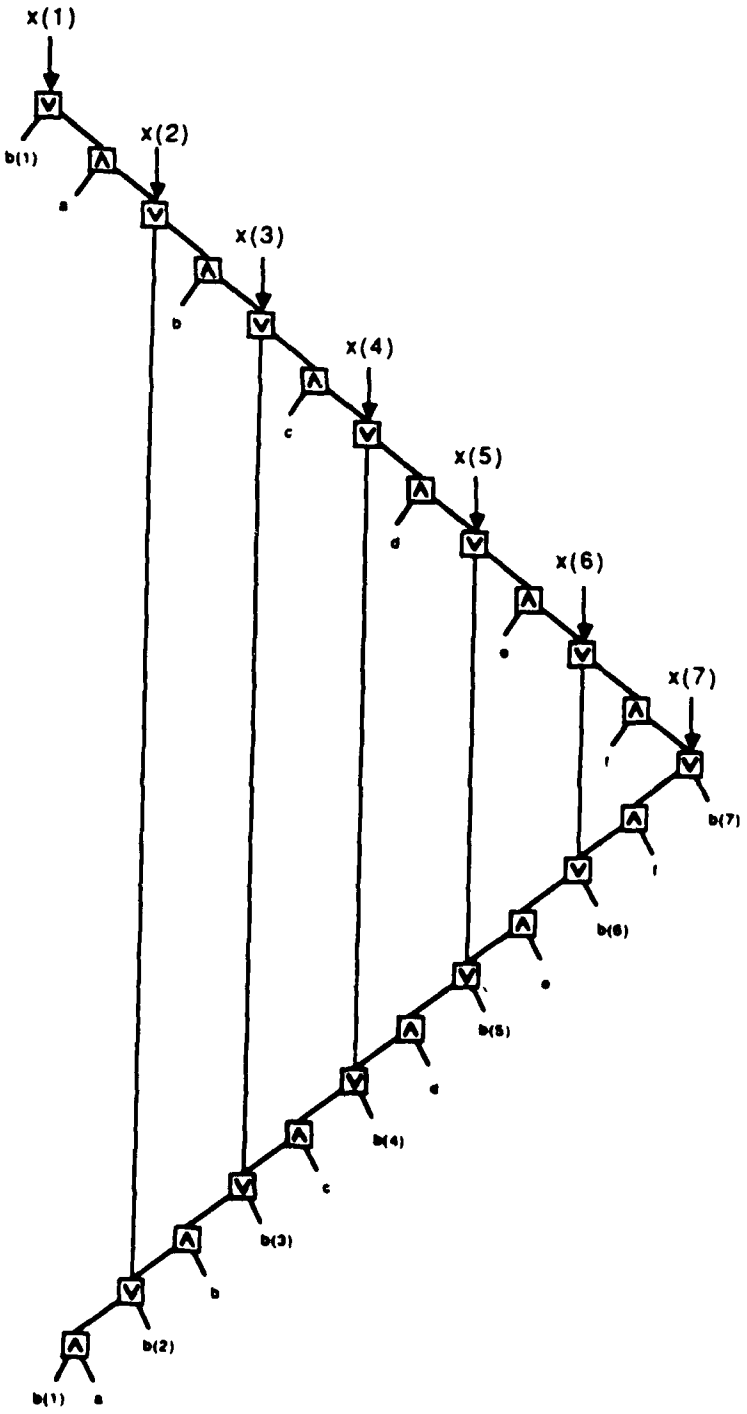


Figure 3: DAG Resulting from Left to Right Elimination Ordering

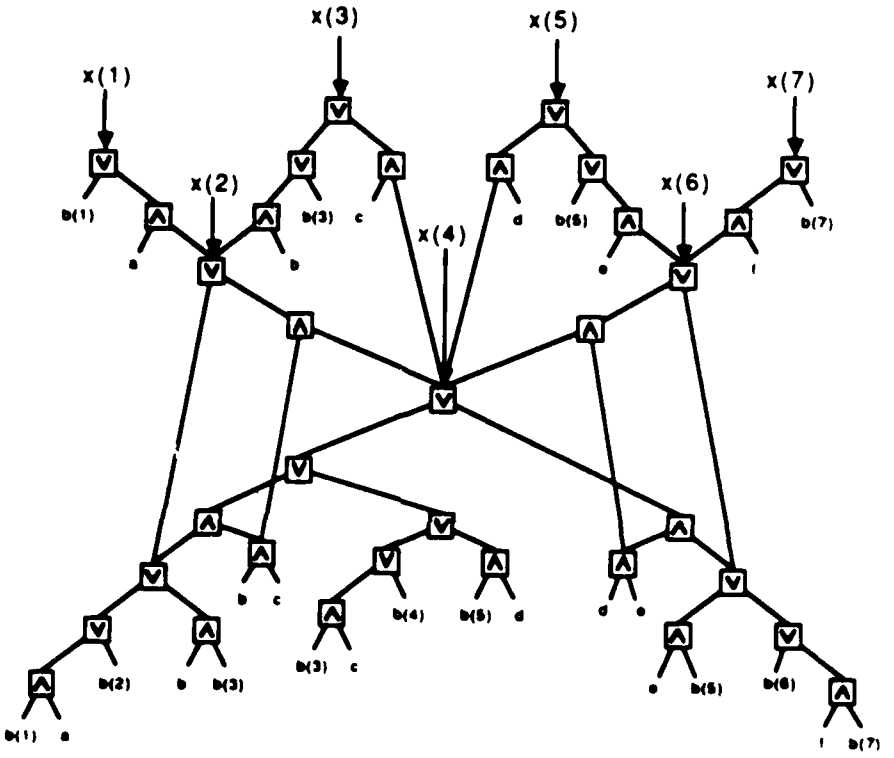


Figure 4: DAG Resulting from Nested Dissection Ordering

The DAGs produced by ANAMOS resemble simple logic networks with AND and OR operations. These expressions can be evaluated on hardware designed for gate-level simulation. If enough processors are used, the DAGS for all subnetworks can be evaluated in parallel and an event-driven scheduler would not be needed. In this scenario, the limiting factor on the amount of time it takes to simulate a clock cycle is the depth of the deepest DAG for all of the subnetworks.

It has already been shown that the structure of DAGs is influenced by the elimination ordering used during Gaussian elimination. The rest of this report is concerned with finding elimination orderings guaranteed to lead to DAGs with small depth. While the discussion is presented largely in terms of Boolean expressions, the results are applicable to other systems solvable by Gaussian elimination.

Chapter 2

Gaussian Elimination

2.1 Introduction

Although Gaussian elimination was originally devised as a technique for solving systems of linear equations, its use has continually been shown to be applicable to problem solving in other areas. Examples are in the solution of path problems, the conversion of finite automata to regular expressions, and the analysis of global flow problems [29,2]. This chapter will discuss the importance of pivot selection in Gaussian elimination and will introduce a graph theoretic approach to the problem.

Systems of equations are represented by the equation

$$Mx = b,$$

where M is an $n \times n$ matrix, x is an $n \times 1$ vector of unknowns, and b is an $n \times 1$ vector of constants. If M is a symmetric positive definite matrix, then it may be factored into the form $M = LDL^T$ where L is a lower triangular matrix and D is a diagonal matrix. Gaussian elimination may be used to find L directly and the equations $Ly = b$, $DL^Tx = y$ may be solved by backsubstitution to find a solution to the system. The total time required to perform the process is $O(n^3)$ in general. However, if M is sparse[1] special provisions may be made to perform elimination

[1]Sparsity will not be formally defined here. It will be considered to mean a matrix, many of whose entries are zero.

$$M = \begin{pmatrix} 12 & 3 & 1 & 7 & \\ 3 & 1 & & 2 & \\ 1 & & 5 & & 2 \\ 7 & 2 & & 5 & \\ & & 2 & & 1 \end{pmatrix} \qquad (PMP^{\mathrm{T}}) = \begin{pmatrix} 1 & 2 & & & 3 \\ 2 & 5 & & & 7 \\ & & 1 & 2 & \\ & & 2 & 5 & 1 \\ 3 & 7 & & 1 & 12 \end{pmatrix}$$

$$\downarrow \qquad\qquad \downarrow$$

$$\begin{pmatrix} 12 & 3 & 1 & 7 & \\ & .25 & -.25 & .25 & \\ & & 4.7 & -.33 & 2 \\ & & & .64 & .14 \\ & & & & .11 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2 & & & 3 \\ & 1 & & & 1 \\ & & 1 & 2 & \\ & & & 1 & 1 \\ & & & & 1 \end{pmatrix}$$

Figure 5: Solution of a System of Equations Under Two Permutations

more efficiently.

The limiting factor in the efficiency of sparse Gaussian elimination is the amount of *fill-in* that occurs during the elimination process. Fill-in is defined as the number of zero entries in the matrix that become nonzero as elimination proceeds. The total amount of fill-in that arises from the solution of a sparse system of equations may be limited to some extent by selecting a proper elimination ordering.

A permutation of the rows and columns of M results in a new matrix that represents the same system of equations as M. In matrix notation, a permutation of the system is defined by

$$(PMP^{\mathrm{T}})(Px) = Pb.$$

where P is an $n \times n$ permutation matrix. The new coefficient matrix PMP^{T} will in general have a different fill-in structure than M. Thus, a matrix may be permuted before Gaussian elimination is performed in order to achieve greater efficiency. An example of a system of linear equations is shown in Figure 5.

Both matrices in the figure represent the same system of equations under different permutations. The upper triangular form DL^{T} results after the final step of the forward elimination phase of Gaussian elimination. Notice that the solution of the equations defined by M has fill-in terms while the solution of PMP^{T} does

not. This figure clearly indicates that savings in computation time may be made by reordering the rows and columns of a matrix.

The permutation of rows and columns need not be explicit; it may be made during the elimination step. An interchange of rows and columns is equivalent to the selection of some arbitrary nonzero diagonal element of M as the pivot. Various methods of pivot selection based on a matrix representation have been presented [22]. These will not be discussed in this report. Instead, their graph theoretic counterparts will be introduced in Section 2.2.

2.2 A Graph Theoretic Approach

Define $G(M)$ to be the graph $G = \langle V, E \rangle$ associated with M, such that each variable in the system of equations is associated with a vertex v_i, $i = 1 \cdots n$, and that for each nonzero entry $M_{i,j}$ there is an edge (v_i, v_j) with head v_j and tail v_i. Such a graph represents the nonzero structure of the matrix M [25]. If M is symmetric, G may be an undirected graph. However, if M is not symmetric, (i.e., $M_{i,j} \neq M_{j,i}$), G will have directed edges.

An elimination ordering is a bijection $\alpha : \{1, 2, \cdots, n\} \leftrightarrow V$ and $G_\alpha = \langle V, E, \alpha \rangle$ is an *ordered graph*. This graph may be used as an aid in selecting an elimination ordering that produces the minimal total fill-in.

Fill-in manifests itself on the graph G as additional edges during the elimination process. Pivoting along a diagonal element in M is equivalent to the removal of a vertex, v, from the graph. The *deficiency* of v, $D(v)$, is the set of edges defined by

$$D(v) = \{(u, w) | (u, v) \in E, (v, w) \in E, (u, w) \notin E\}$$

and represents the set of fill-in edges due to the elimination of vertex v. The graph

$$G_v = (V - \{v\}, E(V - \{v\}) \cup D(v))$$

is called the *v-elimination graph* of G. For an ordered graph, G_α, the elimination process

$$P(G_\alpha) = [G = G_0, G_1, G_2, \cdots, G_{n-1}]$$

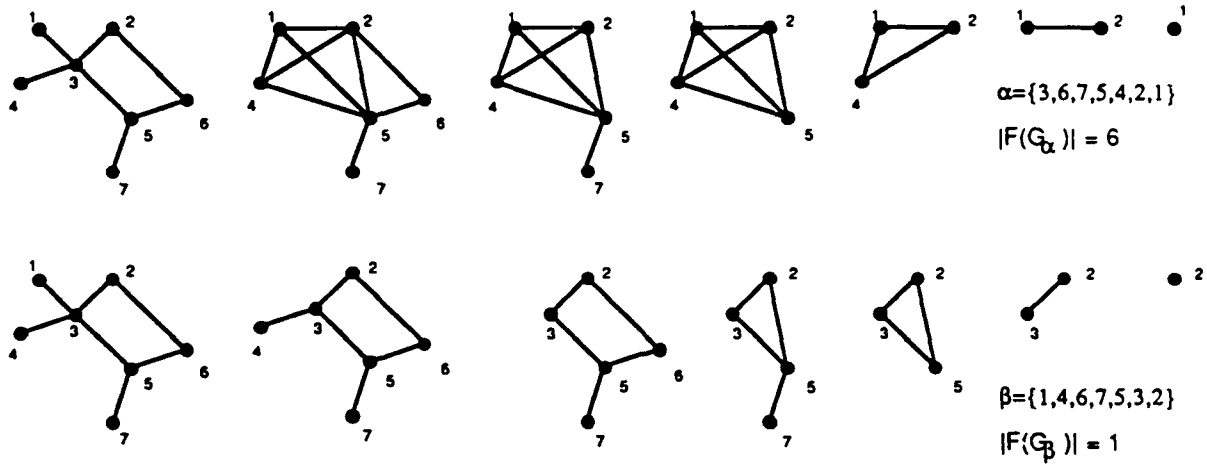


Figure 6: The Fill-in Resulting From Two Different Elimination Orders

is the sequence of graphs that result from the elimination of the vertices in the order specified by α. The total fill-in, $F(G_\alpha)$ is given by

$$F(G_\alpha) = \bigcup_{i=1}^{n-1} D(v_{\alpha_i}).$$

Fill-in that occurs with the elimination of a vertex is a function of the elimination ordering α. An example is shown in Figure 6 which depicts a graph with two elimination orderings, α and β. The elimination steps and the fill-in edges that occur are shown. The total amount of fill-in $|F(G)|$ is labeled for the two elimination orderings. This example illustrates the effect of elimination ordering on the total complexity of the solution of a system by Gaussian elimination. However, finding an elimination ordering that produces the *minimum* fill-in for a given graph is a problem that has been demonstrated to be NP-complete [13,32].

A system that may be solved with no fill-in, (i.e., $F(G_\alpha) = \emptyset$), is called a *monotone transitive graph* or a *perfect elimination graph*. It can be observed that the edges added during Gaussian elimination result in a perfect elimination graph,

$$G_\alpha^* = \langle V, (E \cup F(G_\alpha)) \rangle.$$

Rose termed this the *monotone transitive extension* of a graph and also characterized these graphs as *triangulated graphs* [26]. A triangulated graph is one in which every cycle of length $n \geq 4$ contains a chord. Finding the minimum set of additional edges that results in a triangulated graph has been shown to be equivalent to the problem of finding the best elimination ordering. Even though this observation provides insight into the problem, it leads to no good algorithms for the selection of an elimination ordering [19].

2.3 Path Problems on Graphs

The graph representation of a system of equations is more than merely a tool for minimizing fill-in. The single-source path problem is to find, for all vertices v, an expression $P(s,v)$ that describes all paths from the source s to v. Path expressions are built from the operators $\cup$ (union), $\cdot$ (concatenation), and $*$ (reflexive transitive closure). It has been shown that the solution of this path problem on graphs also serves as a solution to a variety of problems when the path operators are mapped into other domains. These include the solution of systems of linear equations, data flow problems, and shortest path problems [29]. In this paper only those path problems where the desired expressions simply give the conditions under which a path from s to v exists are considered.

In the Boolean algebra of this analysis $\langle B, \vee, \wedge, \mathbf{0}, \mathbf{1} \rangle$, the operations $\vee$ and $\wedge$ denote Boolean "and" and "or," respectively, applied to *functions*, while the distinguished elements **0** and **1** denote the constant functions that yield 0 and 1, respectively, for all argument values. This algebra is sufficient to express path expressions for a simple path analysis. For this application the operators described by Tarjan are mapped as follows: $\cup \rightarrow \vee$ and $\cdot \rightarrow \wedge$. The operator $*$ evaluates to **1** for all Boolean values and will not enter into the discussion herein.

A vertex labeling of graph $G = \langle V, E \rangle$ is an assignment $x(v) \in B$ to each vertex $v \in V$. An edge labeling is an assignment $A(u,v) \in B$ to each edge $(u,v) \in E$. Gaussian elimination may be performed directly on the graph G to solve the linear

system $Ax = b$ when this system is recast as

$$A'x + b = x$$

where $A' = I - A$. The the solution to this system is the same as that for its other form, but the reformulation allows the solution of systems of equations in algebras that do not have an inverse under the concatenation operation $(\cdot)$. In particular, systems of Boolean equations do not have an inverse under $\wedge$.

The Gaussian elimination algorithm described in Figure 7 solves the single-source path problem under Boolean algebra. It begins by placing an initial label $b(v_i)$ representing a Boolean function on each vertex. Each edge (v_i, v_j) is initially labeled with $A_{i,j}$. Graph structure is modified as described earlier, but labels are modified as well. At each step, before a vertex is eliminated, its label is propagated to each of its neighbors. Likewise, the labels of fill-in edges are modified. Backsolving consists of a similar sequence of operations: propagating labels back to vertices that are successively added to the graph. At the termination of the algorithm, each vertex is labeled with an expression $x(v_i)$ that satisfies the single-source path problem as stated above.

The complexity of this algorithm can be analyzed in terms of and-or operations. Define one operation to be of the form $(a \vee (b \wedge c))$. During forward elimination two sequences of operations are performed for each vertex: propagation of values to neighbors and calculation of labels of fill-in edges. If the graph is directed, the total number of operations performed due to forward elimination is

$$\sum_{i=1}^{n} \left[d(v_i) + 2 \sum_{j=1}^{d(v_i)} j \right].$$

Backsubstitution results in the propagation of labels back to vertex v_i from each of its $d(v_i)$ neighbors. Thus, backsubstitution contributes

$$\sum_{i=1}^{n} d(v_i)$$

operations. The total complexity for performing Gaussian elimination on a graph is given by

$$\sum_{i=1}^{n} \left[d(v_i) + 2 \sum_{j=1}^{d(v_i)} j + d(v_i) \right]$$

```
{ Initial Labelling }
for i = 1 to n do
      x(v_i) ← b(v_i)
od

{ Forward Elimination }
V_0 ← V
E_0 ← E
for i ← 1 to (n − 1) do
      V_i ← V_{i−1} − {v_{α_i}}
      E_i ← E_{i−1} ∩ [V_i × V_i]
      for each u ∈ V_i such that (v_{α_i}, u) ∈ E_{i−1} do
            x(u) ← x(v_{α_i}) ∨ [x(v_{α_i}) ∧ A(v_{α_i}, u)]
            for each w ∈ V_i such that (u, v_{α_i}) ∈ E_{i−1} and w ≠ v_{α_i} do
                  if (u, w) ∈ E_i then A(u, w) ← A(u, w) ∨ [A(u, v_{α_i}) ∧ A(v_{α_i}, w)]
                  else E_i ← E_i ∪ {(u, w)}; A(u, w) ← A(u, v_{α_i}) ∧ A(v_{α_i}, w)
                  fi
            od
      od
od

{ Backsubstitution }
for i ← (n − 1) to 1 do
      for each u ∈ V_i such that (u, v_{α_i}) ∈ E_{i−1} do
            x(v_{α_i}) ← x(v_{α_i}) ∨ [x(u) ∧ A(u, v_{α_i})]
      od
od
```

Figure 7: Solution of a Boolean System by Gaussian Elimination

which is more simply expressed as

$$\sum_{i=1}^{n} \left[d(v_i)^2 + d(v_i) \right]. \tag{1}$$

This formula reveals that the total complexity is extremely sensitive to values of $d(v_i)$. If the graph can be operated on in such a manner that $d(v_i)$ is bounded by a constant for all i, then the complexity will be $O(n)$. However, in the worst case, when G is the complete graph of n vertices, K_n, complexity will be $O(n^3)$.

2.4 Choosing an Elimination Ordering

While finding an elimination ordering that results in the minimum total operation count is an NP-complete problem, heuristics that often result in *good* orderings are available [26]. One heuristic based on the results of Equation 1 relates the total complexity of performing Gaussian elimination to the elimination degrees of the vertices. A method called the *minimum degree algorithm* produces an elimination ordering by selecting the vertex with the minimum degree each time the main loop of the elimination process is executed.

This method is computationally efficient and will often find good elimination orderings. For some classes of graphs the minimum degree algorithm results in the elimination ordering that gives the least number of total operations. One example is the class of graph termed *General Series Parallel* (GSP). This class of graph expands on the conventional definition of series-parallel graphs to include graphs containing acyclic branches. Graphs of this class may be constructed inductively by starting with a single vertex and applying the production rules given in Figure 8. Each rule adds a vertex to the graph, as well as one or two edges. Examples of GSP graphs for a complex gate that might arise in an nMOS circuit and a shift network are provided in Figure 9. The shift network channel graph is shown redrawn, to better exhibit its GSP structure.

The importance of GSP graphs is that they arise often in the analysis of MOS networks [7]. Graphs of this type may be eliminated efficiently by finding the reverse of the sequence of production rules that constructed them. It is clear that since each

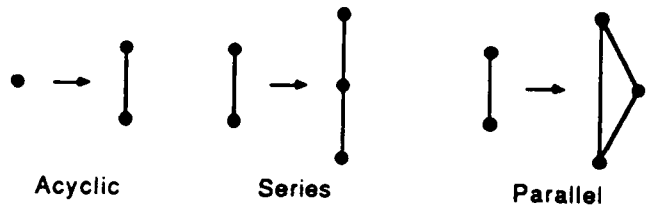


Figure 8: The General Series-Parallel Production Rules

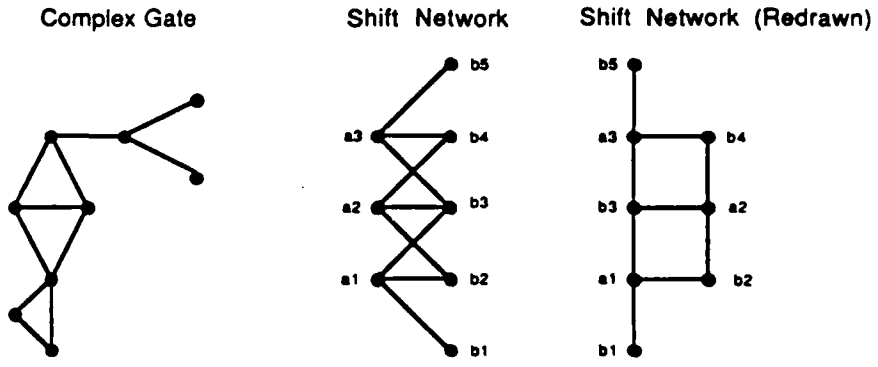


Figure 9: Examples of GSP Graphs

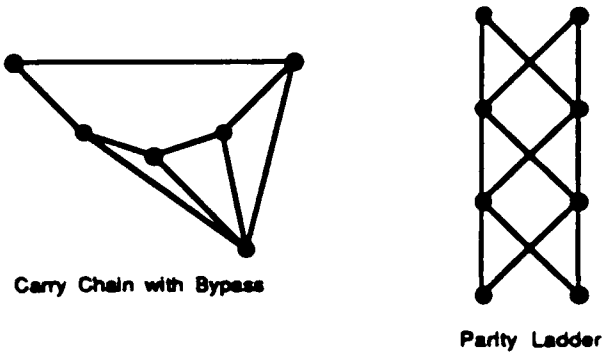


Figure 10: Non-GSP Graphs

production rule adds one vertex adjacent to at most two others, the minimum degree algorithm will effectively find the reverse sequence of production rules applied. The algorithm also results in a low total complexity since at each step an eliminated vertex will have (at most) elimination degree $d(v_i) = 2$. Thus, by Equation 1 the total complexity of solving a system described by a GSP graph will be:

$$\sum_{i=1}^{n} \left[d(v_i)^2 + d(v_i) \right] = 6n \tag{2}$$

Other graphs that arise in the analysis of MOS networks have low elimination degrees even though they may not be GSP. Two examples are shown in Figure 10. These graphs may be eliminated effectively by the minimum degree algorithm with no vertex having $d(v) > 3$. Thus, the minimum degree algorithm leads to low total complexity for these graphs as well.

The minimum degree algorithm is an example of a *greedy* algorithm in that it chooses vertices without regard to future eliminations of vertices. Another greedy algorithm is the *minimum deficiency algorithm*. At each step, a vertex is chosen such that $D(v)$ is minimized. For GSP graphs, the minimum deficiency algorithm will find a reverse sequence of production rules in a manner similar to the minimum

degree algorithm. Each Acyclic and Parallel production rule leads to a vertex with $D(v) = 0$, and each Series rule introduces a vertex with $D(v) = 1$. Thus, at each elimination step, a vertex with $D(v) = 1$ or 0 may be found. In practice, this algorithm often produces an ordering equivalent to the minimum degree algorithm for GSP graphs and has the same bound on elimination complexity.

A drawback of the minimum deficiency algorithm is that it is more computationally intensive than the minimum degree algorithm. Calculating the deficiency of a vertex involves examining all of its pairs of neighbors. Furthermore, if the implementation is not clever, this calculation will be performed for all uneliminated vertices at each step in the elimination process. An advantage of this algorithm over the minimum degree algorithm, however, is that for arbitrary graphs that are triangulated, a perfect elimination ordering will be found [26]. The minimum degree algorithm is not guaranteed to find such an ordering.

This chapter introduced Gaussian elimination and the importance of finding a good elimination ordering. The ordering algorithms examined were greedy in nature and were performed during the elimination process. In Chapter 3, algorithms which produce an ordering before elimination begins and which guarantee good asymptotic complexity for certain classes of graphs will be examined.

Chapter 3

Generalized Nested Dissection

This chapter will discuss a more complicated method of finding an elimination ordering called *nested dissection* which was first proposed for grid graphs that arise in finite element analysis [14]. The algorithm's basic idea is to use a "divide and conquer" strategy on the graph. Removal of a set of vertices results in two new graphs on which Gaussian elimination may be performed separately. The results for the two parts may then be combined to find the solution for the entire graph. This method has been shown to result in good elimination orderings for certain classes of graphs.

An observation about the results produced by nested dissection algorithms is that the path expressions generated in Gaussian elimination tend to exhibit a balanced structure in which a parallel evaluation strategy may be used [23,24]. The importance of balanced expressions was discussed briefly in Chapter 1. Later in this chapter, the structure of expressions will be analyzed in more detail.

3.1 Graph Separators

A *separator* of a graph is a relatively small set of vertices whose removal causes the graph to fall apart into a number of smaller pieces. If S is a class of graph, n is the number of vertices in a graph, and $p(n)$ is some function of n, then S satisfies a *$p(n)$-separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that a separator

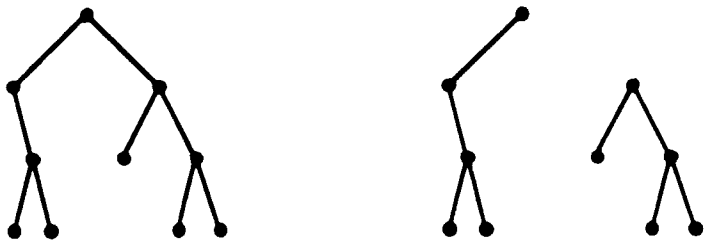


Figure 11: A 1-Separator for a Tree

set with at most $\beta p(n)$ vertices separates the graph into components with at most αn vertices each.

Most algorithms based on separators are recursive, first finding a separator for the whole graph and then finding separators for the components. For these algorithms to work on a graph of class S, all subgraphs of this graph must also be of class S. When a class satisfies this requirement we say that S *is closed under subgraph.*

Binary trees are a class of graph that is closed under subgraph; separation at any vertex separates the graph into two smaller binary trees. Figure 11 shows a binary tree and its decomposition into two smaller trees at a single vertex separator. This theorem is stated for binary trees [16]:

Theorem 1 *The class of binary trees satisfies a 1-separator theorem for* $\alpha = \frac{2}{3}$ *and* $\beta = 1$.

A planar graph is one which can be drawn on a plane so that the edges of the graph only intersect at their endpoints[5]. For planar graphs, the following theorem is taken from Lipton and Tarjan [21].

Theorem 2 *The class of planar graphs satisfies a* $\sqrt{n}$*-separator theorem for* $\alpha = \frac{2}{3}$ *and* $\beta = 2\sqrt{2}$.

In more recent work, Djidjev proved that the theorem also holds for $\beta = \sqrt{6}$ [10].

Series-parallel graphs have long been used to represent simple electric networks [11]. Such graphs may be constructed by the application of series and parallel production rules. These graphs obey the following theorem.

Theorem 3 *The class of series-parallel graphs satisfies a 2-separator theorem for* $\alpha = \frac{2}{3}$ *and* $\beta = 1$.

This is accepted as a folk theorem. We will defer its proof until the next chapter, which is devoted to series-parallel graphs.

These theorems are presented to provide examples of the types of separators that have been shown to exist, and lead to algorithms for finding separators for limited classes of graphs (i.e., binary trees, planar graphs). Except for the simplest cases, finding separators is a non-trivial problem and no good algorithms exist for finding separators greater than two in size for arbitrary graphs.

3.2 Elimination Ordering Algorithms

Many variations of elimination ordering algorithms are based on nested dissection. These algorithms have the following basis as a common starting point. The main differences involve the separators found for different classes of graphs and the resulting complexity bounds.

Given a graph G with n vertices, partition the graph into parts C, A_1, A_2, etc., such that C is a separator of the graph. Number the vertices in C from n down to $(n - |C| + 1)$ so that they are eliminated last from the graph. Recursively number the elements of each of the remaining parts of G, $(A_1, A_2, \cdots)$ from 1 to $(n - |C|)$. The procedure continues until all vertices are numbered. Typically, the recursion will cease when the size of a set reaches some small threshold value, n_0, in which case the vertices of the set are arbitrarily assigned numbers in the given range.

3.2.1 Nested Dissection

Alan George proposed the first nested dissection algorithm [14]. It was defined only for grid graphs for which there are simple separators. Figure 12 shows a grid graph

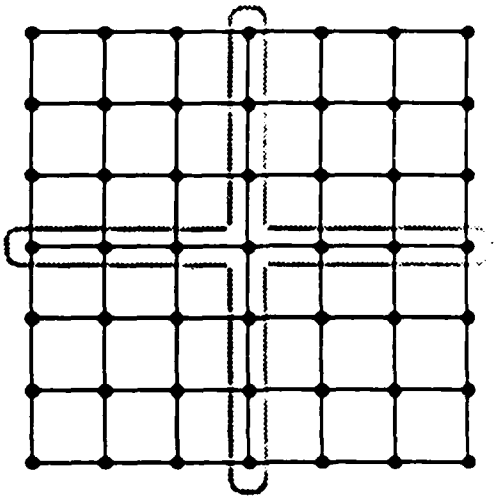


Figure 12: A Grid Graph of Size $n = k \times k$ With Separator Set Indicated

with $n = k \times k$. Removal of the middle column and middle row separates the graph into four subgraphs. The algorithm is as follows. Assume that k is one less than a power of two.

- Remove row $(k+1)/2$ and column $(k+1)/2$. Give the highest numbers to these $2k-1$ vertices.
- There are now four components of the original graph. If their sizes are greater than one, recursively number the components. Otherwise, number the four vertices in the range specified.

Graphs where k is not one less than a power of two may be handled by adding some number of "dummy" vertices. This algorithm is well suited for finite element mesh analysis and results in $O(n \lg n)$ fill-in and $O(n^{\frac{3}{2}})$ total operation count.

3.2.2 Automatic Nested Dissection

This algorithm is defined for any arbitrarily connected graph [15]. A graph may be partitioned into *levels* by performing a breadth-first search beginning at some starting vertex v. Each level, $L_0, \ldots, L_r$ is a partition of the graph. L_0 contains vertex v, L_1 contains those vertices adjacent to v, L_2 contains those vertices two edges away from v, etc. Each level is, to some extent, a separator of the graph. The algorithm is as follows:

- Partition the graph into levels $L_0, \ldots, L_r$ by performing a breadth-first search on the graph.
- The vertices of level $s = \lfloor (r+1)/2 \rfloor$ separate the graph. Choose a minimal subset of L_s that is still a separator and assign the highest numbers to these vertices.
- Recursively number each component (there may be more than two) whose size is greater than n_0.

It has been suggested that this algorithm results in $O(n \lg n)$ fill-in for a number of finite element meshes. While it may have asymptotic complexity approaching this limit, we have observed better performance from the minimum degree algorithm described in Section 2.4. Table 1 shows the results we have obtained for a variety of square grid graphs of size $\sqrt{n} \times \sqrt{n}$.

3.2.3 Generalized Nested Dissection - I

This algorithm was derived by Lipton, Rose and Tarjan [20]. Given a graph G of class S that obeys an $f(n)$-separator theorem with constants α and β, partition the graph into three parts A, B, and C, such that C is a separator of the graph with no more than $\beta f(n)$ vertices.

- If there are no more than $n_0 = (\frac{\beta}{1-\alpha})^2$ vertices, number them arbitrarily in the range specified.

Grid Size	Minimum Degree Algorithm		Automatic Nested Dissection	
(n)	fill-in	complexity	fill-in	complexity
16	18	172	22	200
25	37	350	46	420
36	71	678	100	1032
49	122	1228	165	1824
64	178	1840	240	2822
81	280	3198	321	4010
100	376	4442	454	5996
121	487	6078	612	8556
144	649	8662	819	12086

Table 1: Comparison of Minimum Degree and Automatic Nested Dissection

- Find sets A, B, and C that satisfy the $\sqrt{n}$-separator theorem where C is the separator set.
- Assign the vertices of C the highest numbers.
- Delete all edges whose endpoints are both in C. Apply the algorithm recursively to the subgraph induced by $B \cup C$ and also to the subgraph induced by $A \cup C$.

The vertices of C are included in the recursive call; therefore some method of recording which vertices have already been numbered is required. This algorithm was analyzed in particular detail for classes of graph that follow a $\sqrt{n}$-separator theorem. From a result of Lipton and Tarjan we know that planar graphs satisfy a $\sqrt{n}$-separator theorem [21]. The ordering produced by this algorithm will result in $O(n \lg n)$ fill-in and $O(n^{\frac{3}{2}})$ total operation count for planar graphs, although the coefficients of the actual fill-in and operation counts are very large. However, the authors believe that their worst-case bounds are very pessimistic and that the algorithm would be useful for very large graphs. Applications of such a theorem include finite element meshes (which are planar embeddings of a planar graph) and GSP graphs, which also belong to the class of planar graphs.

3.2.4 Generalized Nested Dissection - II

A variation to the generalized nested dissection algorithm of the preceding section has been proposed for separators that divide the graph into more than two pieces, A and B [16]. This algorithm assumes that the separator C splits the graph into pieces $A_1, A_2, \cdots, A_r$. A separate recursive call is made for each part, A_i, $1 \leq i \leq r$.

- If there are no more than n_0 vertices, then simply number the vertices arbitrarily in the range given.
- Find a separator with $k \leq \beta\sqrt{n}$ vertices that divides the graph into pieces $A_1, A_2, \cdots, A_r$, where $|A_i| \leq \alpha n$. Number the vertices of C arbitrarily from $(n - |C| + 1)$ to n.
- Call the algorithm recursively r times for each component A_i, $1 \leq i \leq r$ to number the remaining vertices in the range between 1 and $n - |C|$.

This algorithm differs from the previous one in that it does not include the vertices of C in the recursive call. Also, by recursing on more than two subgraphs at each level it does not, in general, result in the same bounds for fill-in and total operation count. However, for planar graphs, the same bounds are met. For other classes of graph with $\sqrt{n}$-separator theorems it may even perform better [16]. For the rest of this report, the term "generalized nested dissection" will be considered to refer to this variation.

3.3 Separator Trees

The recursion of the algorithms described above suggests a natural decomposition of graphs in terms of their separators. At the highest level is a separator that divides the graph into components. These components themselves have separators, and so on. At the lowest level are components that may be divided no further, possibly containing only a single vertex. This decomposition may be described in terms of a structure called a *separator tree*.

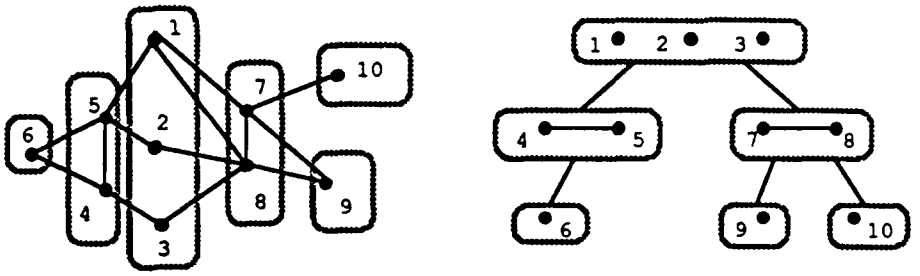


Figure 13: A Graph and Its Separator Tree

A separator tree for a graph is shown in Figure 13. Such trees graphically show how separators arise from a graph; they also reveal where fill-in may occur. The following theorem is from [27].

Theorem 4 *Let $G = \langle V, E, \alpha \rangle$ be an ordered graph. Then (v, w) is an edge of G_α^* if and only if there exists a path $\mu = [v = v_1, v_2, \cdots, v_{k+1} = w]$ in G_α such that*

$$\alpha^{-1}(v_i) < min(\alpha^{-1}(v), \alpha^{-1}(w)) \text{ for } 2 \leq i \leq k.$$

This theorem states that an edge (v, w) fills in if and only if there is a path from v to w containing only vertices eliminated before either v or w.

Theorem 4 may be used to calculate bounds on fill-in due to a nested dissection algorithm. Consider a node of the separator tree C, and its subtrees $A_1, A_2, \cdots, A_n$. Since there are no paths between A_i and A_j initially (for any two subtrees where $i \neq j$), and the elements of C are given higher elimination numbers than those in A_i and A_j, there may not be a fill-in edge between any member of A_i and A_j. Thus, the separator tree shows that the only possible fill-in that may occur is along the edges of the tree, or between the vertices of an individual node of the tree. This fact may be used to calculate bounds on the total amount of fill-in using a nested dissection ordering algorithm for some classes of graph.

3.4 Complexity Versus Structure

Nested dissection algorithms are based on a divide-and-conquer strategy that successively splits the graph into smaller subproblems to be solved independently. Information about the smaller problems is then combined to find the solution of the whole. The discussion on separator trees showed how this technique could lead to bounds on the amount of fill-in that occurs and also bounds on total complexity.

The divide-and-conquer approach also leads to path expressions that exhibit balance. Gaussian elimination can be viewed as the propagation of information through a graph. Path expressions are built by propagating information about the graph to one point and then propagating the information back to all points. The structure of the expressions generated depends on how the information flows through the graph, which is determined by the elimination ordering.

In Chapter 1, a simple graph as shown in Figure 2 was eliminated with two different orderings of vertices. Elimination of vertices from left to right resulted in the DAG of Figure 3, while a different elimination ordering resulted in a more balanced DAG, that of Figure 4. The elimination ordering of the second can be seen to be one that would be produced by a generalized nested dissection algorithm, eliminating the middle vertex (a separator) last.

A graph resembling a binary tree is shown in Figure 14. The minimum degree algorithm could result in a number of different elimination orderings, one of them as shown in Figure 15. The resulting DAG is shown as well. Once again, this DAG is not balanced, but is "long and skinny." A generalized nested dissection algorithm would guarantee elimination of the vertices in an order similar to that depicted in Figure 16. The resulting DAG for this ordering is shown as well. It is obvious that this DAG is more balanced, just as the DAG of Figure 4. We can say that this DAG is "short and fat."

In many situations, these path expressions of the DAG are calculated symbolically (as shown) and used repeatedly to solve problems that involve different labelings of graphs with the same structure. A long, skinny expression tree means that all operations must be performed sequentially, whereas a more balanced tree shows a

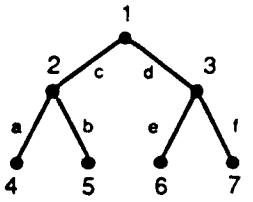


Figure 14: A Graph Resembling a Binary Tree

level of independence among the operations. An appropriate architecture may take advantage of this independence by using multiple processors to exploit fine-grained parallelism.

In general, the expressions generated by Gaussian elimination using nested dissection ordering algorithms are balanced, and hence have small *depth*, if there are good separators for the graph. In the following chapters algorithms that lead to the identification of separators for a variety of graphs will be presented. The extent to which they separate the graph will be shown to lead to an upper bound on the depth of expressions generated.

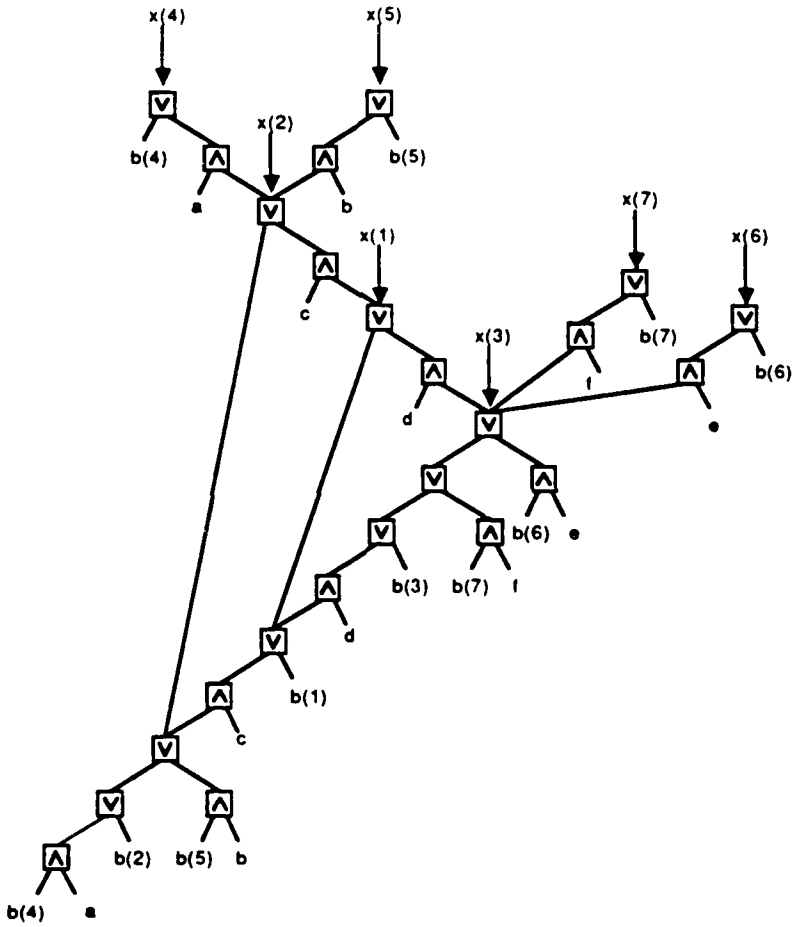


Figure 15: The Resulting DAG With an Ordering Due to the Minimum Degree Algorithm

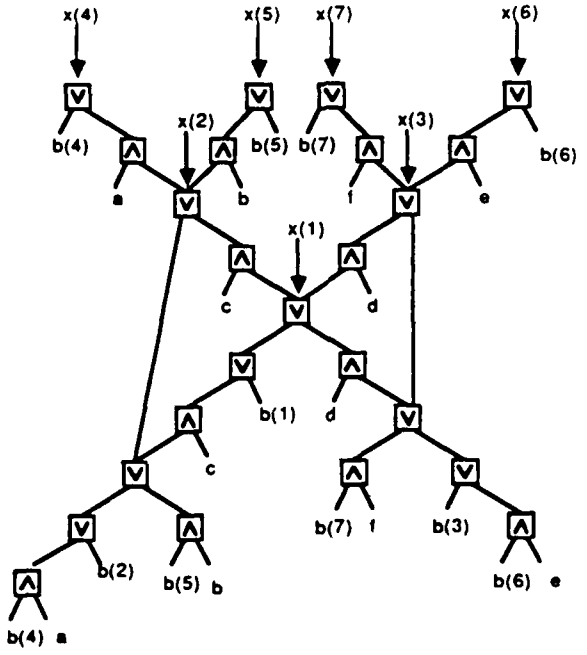


Figure 16: The Resulting DAG With an Ordering Due to Nested Dissection

Chapter 4

Separators for Series-Parallel Graphs

In this chapter some properties of a class of graph called *Series-Parallel* (SP) are examined.. This class of graph is often used to describe electrical networks, since many networks are composed of only series and parallel connections of elements. Reduction rules for SP graphs are developed first. It is then shown how these can be used to find separators for these graphs.

4.1 Series-Parallel Reduction Rules

SP graphs may be described by the composition of their edges in terms of Series, S, and Parallel, P, reduction rules. The two rules are shown in Figure 17. Each of the rules results in the reduction of the number of edges in the graph by one. By the application of a sequence of these reduction rules, it is possible to reduce a SP graph to a single edge. For SP graphs the following applies.

Theorem 5 *A multigraph is SP if and only if it can be reduced to the trivial SP graph (two vertices joined by a single edge) by a sequence of series and parallel reductions.*

This is a trivial generalization of a corollary proposed by Duffin [11]. Thus, the application of these reduction rules may be used to test if a graph is indeed SP.

Figure 17: Series and Parallel Reduction Rules

There may be many different combinations of the rules that result in a single edge, all of which being valid reductions of the graph. This particular set of reduction rules has been shown to possess the *Church-Rosser property* [30] which means that the order in which the rules are applied will not affect the possiblity of reducing the graph to a single edge.

The manner in which the reduction rules are applied to a graph can be described by a *binary decomposition tree* [31]. Such a tree is analagous to a binary expression tree, where the operators represent functions performed on the edges of the graph. Figure 18 illustrates the construction of a decomposition tree for a simple SP graph where endpoints are used to identify the edges. A decomposition tree may be built by labeling the edges of the graph for each reduction where the labeling procedure is described in Figure 19. The initial labels of an edge are assumed to be the set of its endpoints.

It is interesting to note that if the **P** and **S** operators are interpreted as the Boolean operators $\vee$ and $\wedge$, the decomposition tree gives all paths between the two remaining vertices in much the same manner that the expressions generated by Gaussian elimination describe paths for all pair of vertices. In fact, the reduction procedure described is identical to the forward elmination step of Gaussian elimination where vertices are intially labeled with $x(v_i) = 0$ and the ordering algorithm used is simply the selection of some vertex with $d(v_i) = 2$.

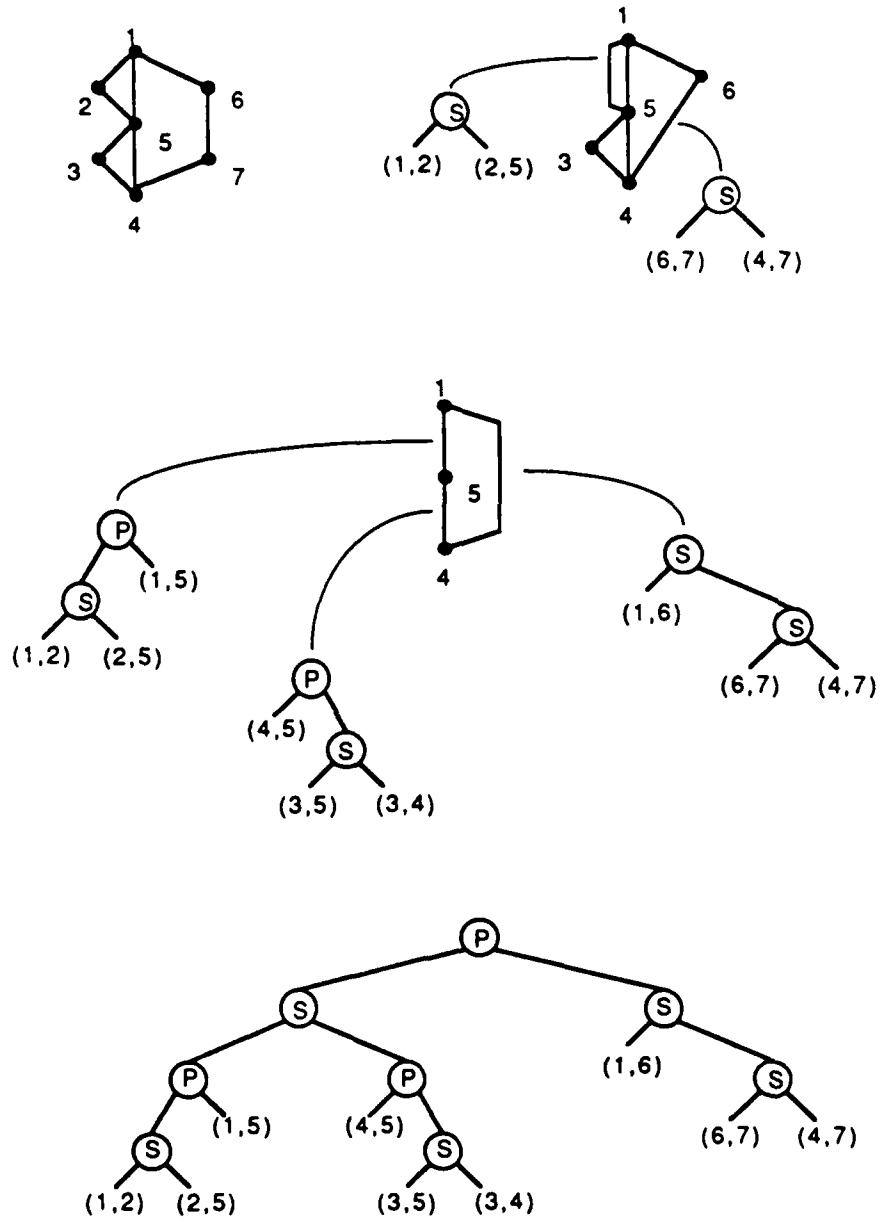


Figure 18: Construction of a Decomposition Tree for a SP graph

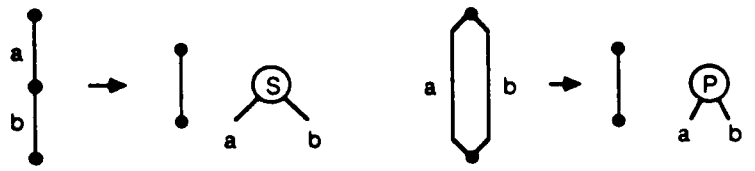


Figure 19: Edge Labeling for the Reduction of SP Graphs

4.2 Separators of SP Graphs

Earlier a theorem that SP graphs have 2-separators for $\alpha = \frac{2}{3}$ was provided (Theorem 3). In this section it is shown that, with a proper labeling of the decomposition tree of a SP graph, such separators may be easily found.

Each S node in the decomposition tree represents the removal of one vertex from the graph. Each S node, n, will be assigned an internal weight $i(n) = 1$ to reflect this. Conversely, a P rule represented the removal of no vertex, but only the removal of an edge. Each P node will be given an internal weight $i(n) = 0$.

Each node n in the tree, either S or P, represents a separator of the graph. Its total weight

$$w(n) = \sum_{p \in descendants(n)} i(p)$$

gives the number of vertices separated from the rest of the graph by this separator. A mapping between nodes in the tree and separators may be easily constructed by the proper labeling of each node in the tree with an additional field. Each leaf node represents an edge, (v_1, v_2). A Series combination of two SP subgraphs defined by their endpoints (v_1, v_2) and (v_2, v_3), results in the separator set (v_1, v_3) that serves to separate everything internal to (v_1, v_3) from the rest of the graph. Parallel rules always combine two subgraphs whose endpoints are the same. By labeling each node of the tree with the endpoints of the subgraphs they represent, each node will be labeled with a separator pair. Figure 20 shows the labeling operations for series and parallel rules. By applying these from the bottom up each node in a decomposition tree may be labeled with its separator set and total weight. The tree

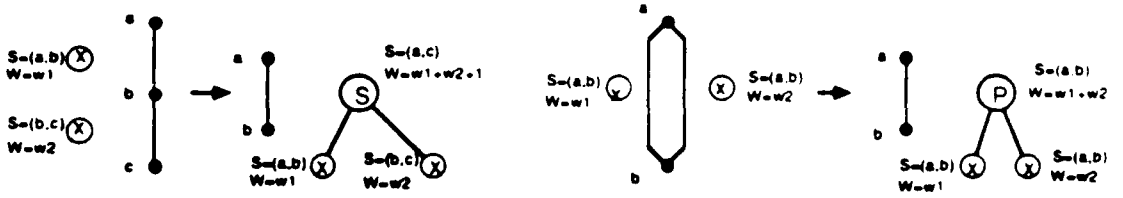


Figure 20: Labeling for Each Node in a Decomposition Tree

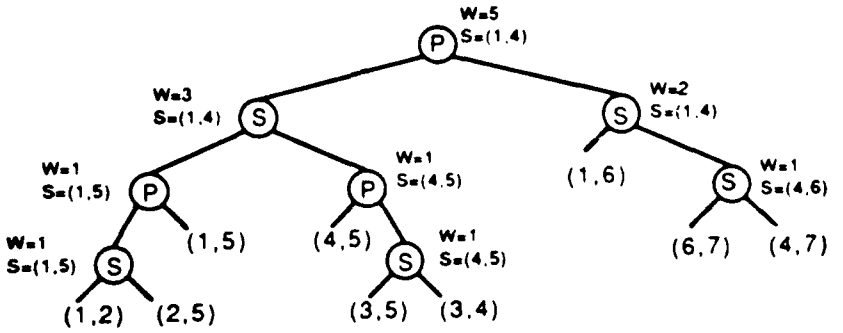


Figure 21: A Decomposition Tree After Labeling

of Figure 18 is shown again in Figure 21 with each node labeled with the proper total weight w and its separator set S.

It should be noted that the weight of the root of the decomposition tree is two less than the total number of vertices in the graph. This is because the weight of each node represents how many vertices would be separated from the rest of the graph, not including the two vertices of the separator set. The algorithm of Figure 22 is a direct implementation of the general step of the constructive proof and finds a 2-separator in a graph for which a decomposition tree has been constructed. In the next section this function will be used as part of another algorithm for computing

```
function Find2Separator( tree T, weight W) returns tree
    if w(T) ≤ 2W/3 then return T
    else
        if w(LSON(T)) ≤ w(RSON(T)) then
            Find2Separator(RSON(T), W)
        else
            Find2Separator(LSON(T))
        fi
    fi
end Find2Separator;
```

Figure 22: Recursive Algorithm for Finding 2-Separator of a SP Graph

an elimination ordering for a SP graph based on nested dissection. At this point the theorem that SP graphs have 2-separators with $\alpha = \frac{2}{3}$ will be proved.

PROOF: Nodes are examined in the decomposition tree starting at the root and descending through the tree. The general step of the proof will eventually accept a node as a representing a valid separator of the SP graph. W represents not the total weight of the tree, but the number of nodes in the graph. For each node in the tree there is a separator set of size two that separates the vertices of its descendent rules from all others in the graph. It is obvious that some node, $n1$, whose weight is less than or equal to $\frac{2}{3}W$ may be found but it remains to be shown that the weight of everything else in the tree, $(W - [w(n1) + 2])$, is also $\leq \frac{2}{3}W$.

INITIALIZATION: n1 is the root of the tree.

INVARIANT: $W - (w(n1) + 2) \leq \frac{2}{3}W$. This is initially true when $n1$ is the root since $W = w(n1) + 2$.

GENERAL STEP: Find the child of $n1$ with the greatest weight. Call this node $n2$. If $w(n2) > \frac{2}{3}W$ then repeat the general step with $n1 \leftarrow n2$. The invariant still holds.

Otherwise, $n2$ is a node that represents a separator set of size two that divides the graph as required. The weight of $n2$, $w(n2)$, is small enough; it remains to be shown that the weight of the rest of the graph without the separator set is small enough as well.

It is known that $w(n1) > \frac{2}{3}W$. Also, since each node in the tree has two children and (at most) internal weight 1, $w(n2) \times 2 + 1 \geq w(n1)$. Therefore,

$$w(n2) > \frac{1}{3}W - \frac{1}{2}.$$

The number of vertices in the rest of the graph is given by $(W - [w(n2) + 2])$. Using the previous relation we get

$$(W - [w(n2) + 2]) \leq \frac{2}{3}W - \frac{3}{2}$$

which indicates that both parts are small enough.

□

4.3 An Elimination Ordering for SP Graphs

The method of finding separators for SP graphs discussed above, can be used to produce an elimination ordering using a procedure based on generalized nested dissection. While SP graphs can be eliminated efficiently using the minimum degree algorithm, we can expect the nested dissection elimination ordering to yield expressions that will exhibit balance, and whose *depth* will be small.

An obvious implementation of a nested dissection algorithm would proceed as follows:

- If there are only one or two vertices, number these in the range specified.
- Build a decomposition tree for the SP graph.
- Find a separator for the SP graph with decomposition tree given by the algorithm of Figure 22.
- Number these two vertices the highest.

- Elimination of the separators results in two new SP graphs. Recursively number these.

This would not be very efficient however, since it would require the building of a decomposition tree for each subgraph. A key observation of the decomposition tree is that the removal of a node splits the tree into two trees that describe the subgraphs of the original graph defined by the separator. The recursive algorithm presented in Figure 23 produces an equivalent elimination ordering by relabeling fields of the modified decomposition tree instead of rebuilding a tree each time.

The algorithm assumes that a decomposition tree has been produced and correctly labeled as described earlier. The variable n is assumed to contain the total number of vertices in the graph, and global variables *ElimCnt* and *ElimNums* are used by the procedure *AssignNumber* for the numbering of vertices. Another function called *Label* is responsible for recalculating the weights of the entire tree when a subtree has been removed.

Removal of a node from the tree may be reflected by simply marking its weight field, w, to zero. The procedure *Label* does not descend past nodes with zero weight. Upon completion of the algorithm, the vertices of the graph will be numbered in an order that represents nested dissection for a SP graph.

Applying the elimination ordering produced by the algorithm of Figure 23 to Gaussian elimination to a system of Boolean equations represented by the graph of Figure 24, the DAG of Figure 25 is obtained. For comparison, the results of the minimum degree algorithm are shown as well in Figure 26. As expected, the generalized nested dissection algorithm for SP graphs yields expressions with a small depth (13), while the minimum degree algorithm resulted in expressions of depth 20.

The algorithm proposed for finding separators of SP graphs is interesting in that it closely resembles the forward elimination part of Gaussian elimination. In the next chapter, this technique will be expanded to handle graphs that are not SP.

```
type nodetype = (Series,Parallel);

type tree = pointer to record
      verts s1, s2;          {The vertices of the separator set}
      int w;                 { The weight of the node}
      tree LSON, RSON;       {The two children of this node}
      nodetype rule;         {Whether this node represents a Series or Parallel rule}
endrecord;

{Global variables}
int ElimCnt =n;
array of int ElimNums[1..n] = {0,0,···,0};

procedure AssignNumber(vert v)
      if ElimNums[v] = 0 then
            ElimNums[v] ← ElimCnt;
            ElimCnt ← ElimCnt − 1;
      fi
end AssignNumber;

function Label (tree T) returns int
      if T.w = 0 then return 0;
      else
            T.w = Label(T.LSON) + Label(T.RSON);
            if T.rule = Series then T.wT.w + 1; fi
            return T.w;
      fi
end Label;

procedure ComputeOrdering( tree T)
      if T.w = 0 then
            AssignNumber(T.s1);
            AssignNumber(T.s2);
      else
            t ← Find2Separator(T);
            AssignNumber(t.s1);
            AssignNumber(t.s2);
            t.w ← 0;
            Label(T);
            ComputeOrdering(T);
            ComputeOrdering(t);
      fi
end ComputeOrdering;
```

Figure 23: Nested Dissection Elimination Ordering Algorithm for SP Graphs

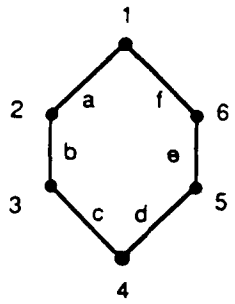


Figure 24: A Sample Graph

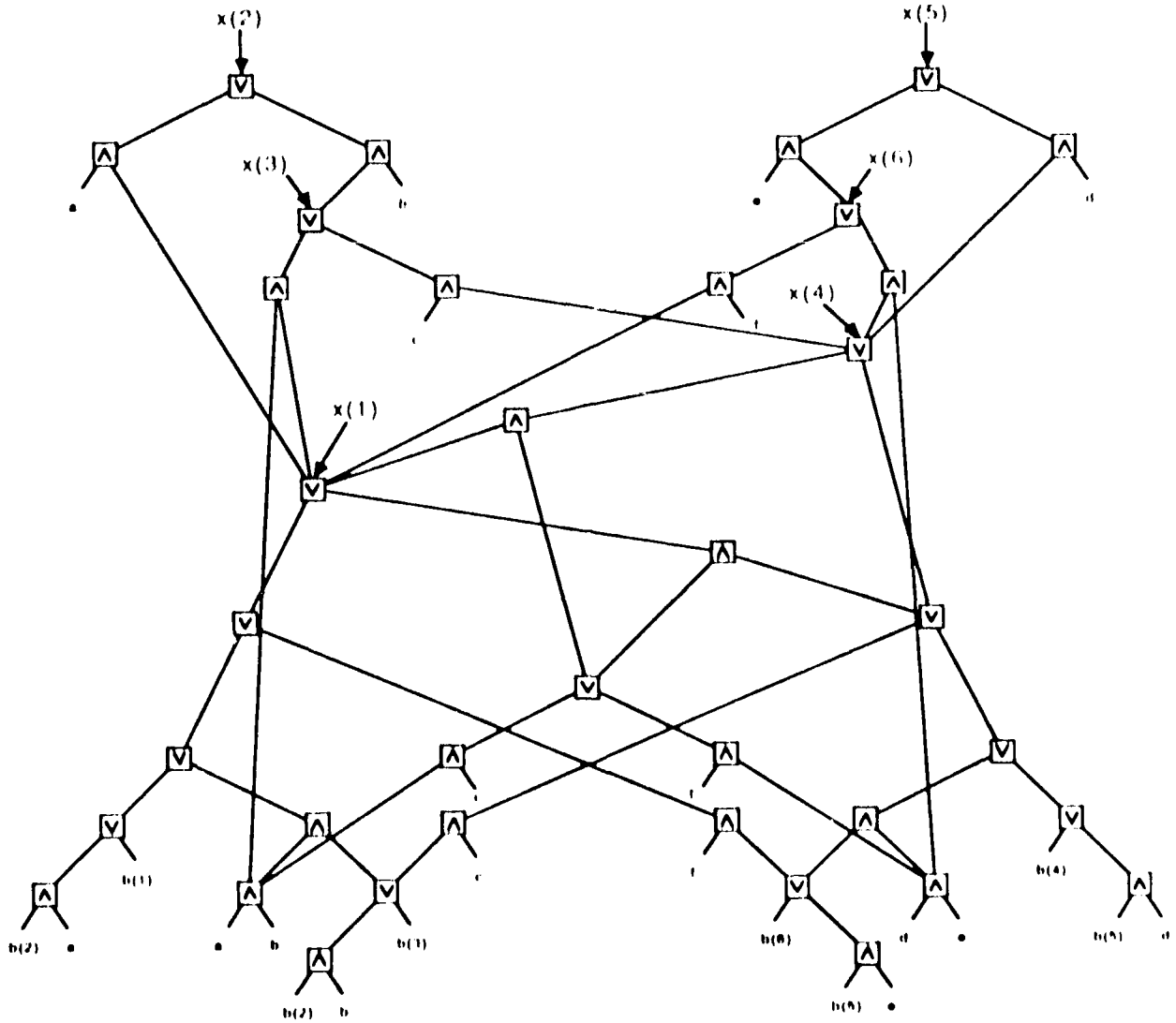


Figure 25: The DAG Resulting from the Generalized Nested Dissection Algorithm

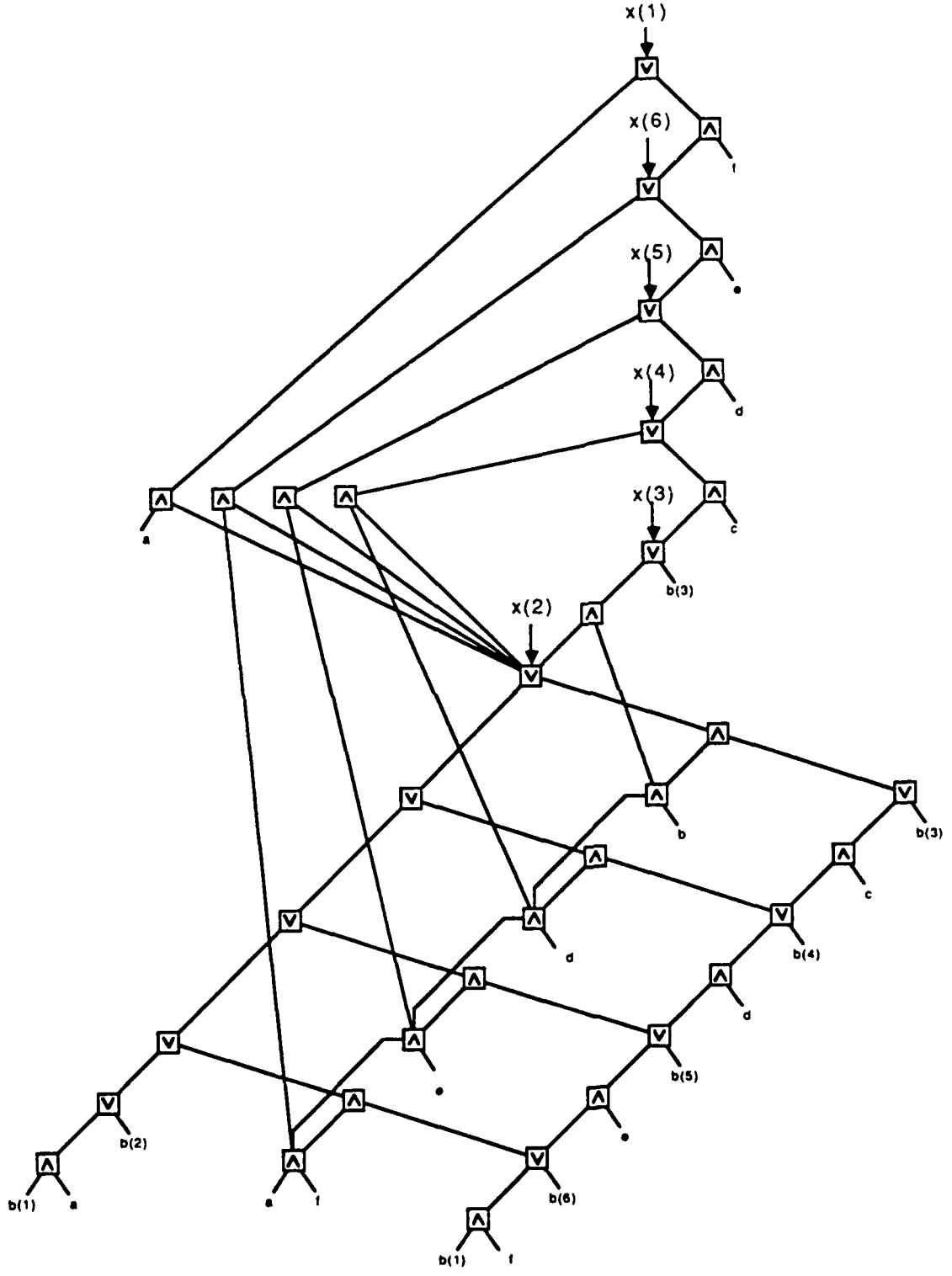


Figure 26: The DAG Resulting from the Minimum Degree Algorithm

Chapter 5

Separators for Arbitrary Graphs

The identification of separators in arbitrary graphs is a difficult problem. While there are good algorithms for finding bi-connected [3] and tri-connected [18] components in graphs that lead to the identificaton of 1- and 2-separators, and algorithms that find separators for certain classes of graph, (e.g., planar graphs), there are no good algorithms for finding separators in arbitrary graphs. Identification of only the 1- and 2-separators in a graph is not sufficient, and arbitrary graphs are not guaranteed to have them anyway. A more general method of finding separators for use with a nested dissection ordering algorithm is desired.

This chapter will develop a method for finding separators in arbitrary graphs based on decomposition trees. Reduction rules will be formulated based on the elimination of vertices that occurs in Gaussian elimination. Using an elimination ordering that gives expressions of small total complexity, it will be shown that the elimination sequence can be reordered to obtain expressions of small depth.

5.1 Elimination Cliques

In the previous chapter, an algorithm was presented for finding 2-separators in SP graphs. The reduction rules proposed for SP graphs repeatedly modified the graph until a single edge remained. It was shown that the resulting decomposition tree for SP graphs gave information about paths through the graphs, and that in this

respect, was very similar to the result of Gaussian elimination for Boolean systems. In fact, the Series reduction rule is exactly the process that occurs in Gaussian elimination when a vertex of degree two is removed: one vertex is removed and a fill-in edge is added to the graph.

An important observation of the information presented in Chapter 4 is that the endpoints of each edge and fill-in edge represented a separator of the graph. In the elimination of SP graphs, each edge that fills in does so to maintain paths that existed between two vertices before the elimination of a vertex. Theorem 4 stated that an edge (v, w) fills in only if there is a path from v to w containing only vertices eliminated before either v or w. The endpoints of such an edge separate all vertices comprising the path from the rest of the graph. Since an eliminated vertex in an SP graph may only contribute to one fill-in edge, it is possible to record the vertices in the paths defined by a fill-in edge by using an appropriately labeled decomposition tree for each edge in the graph.

Each step in the forward elimination phase of Gaussian elimination results in the removal of a single vertex, v_i, from the graph. For SP graphs, a vertex has exactly two neighbors at the time of its elimination. The basic operation is the same whether there is an edge between the neighbors of v_i or not. This one rule is illustrated in Figure 27 and indicates an optional edge between two vertices. If there is an edge, elimination of v_i does not result in a fill-in edge, but merely a labeling of the edge with a new value. If there is not an edge, elimination of v_i results in a true fill-in edge. In either situation, the set of the neighbors of v_i is a 2-separator of the graph, separating one or more vertices from all others in the graph. For the purpose of finding separators, a single reduction rule may be defined: the SP reduction rule. The application of this *one* rule involves merely finding any vertex with two neighbors.

In an arbitrary graph, there may be more than two neighbors of v_i. However, the set $adj(v_i)$ still serves to separate v_i from the graph. Because the number of neighbors of v_i may be greater than two, the result of elimination of v_i is not, in general, a single fill-in edge, but is the set of edges

$$C(v_i) = \{(u, w) | u \in adj(v_i), w \in adj(v_i), u \neq w\}.$$

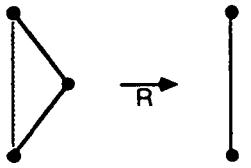


Figure 27: A Single Reduction Rule for SP graphs

This is similar to the deficiency of a vertex, but includes edges already present in the graph as well. The set of vertices, $adj(v_i)$, and edges, $C(v_i)$, form a clique[1] in the graph, G_i, resulting after elimination of v_i. The edges represent sections of all paths of which v_i was a member. In SP graphs, there was exactly one of these edges. However, in an arbitrary graph, v_i is a member of all of the paths defined by the edges $C(v_i)$.

Elimination of another vertex, $v_{i+1} \in adj(v_i)$, results in another clique in the graph G_{i+1}. The edges of this clique contain information about v_{i+1} and v_i, since

$$(adj(v_i) - \{v_{i+1}\}) \subseteq adj(v_{i+1})$$

and paths between members of $adj(v_{i+1})$ through previously eliminated vertices include paths through $adj(v_i)$. This can be stated as the following:

Theorem 6 *In the elimination process, the vertices defined by $adj(v_i)$ separate v_i from the rest of the graph. If $v_j \in adj(v_i)$ and $j > i$, then $adj(v_j)$ also separates v_i from the rest of the graph.*

The structure appearing in this process is the *elimination clique* and its boundary vertices. In SP graphs, this was a single edge, and a labeling scheme could simply use edges to keep track of information concerning the constituent vertices of a path. However, the arbitrary sized elimination clique suggests no such simple labeling scheme.

---

[1] This clique is sometimes referred to as an *elimination clique*.

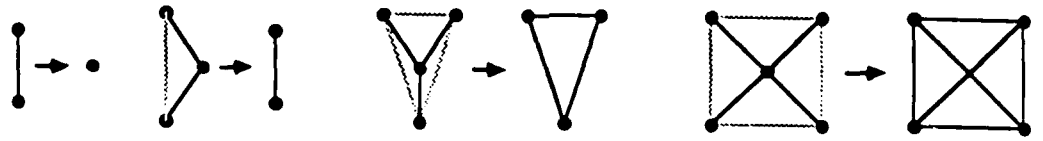


Figure 28: Reduction Rules for Arbitrary Graphs

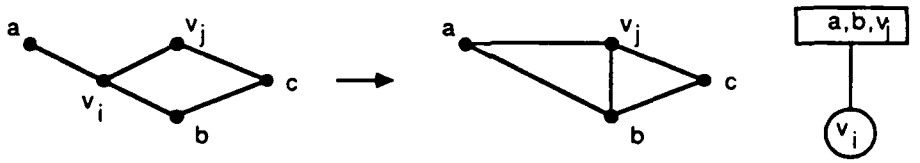


Figure 29: A Decomposition Tree Fragment

5.2 Reduction Rules for Arbitrary Graphs

A simple set of reduction rules for arbitrary graphs can be formulated in terms of the Gaussian elimination procedure discussed. Each rule removes a single vertex from the graph and adds fill-in edges to keep all existing paths present. The new reduction rules have as their single criterion for application the number of neighbors of vertex v_i. Figure 28 shows a few of the reduction rules for small numbers of neighbors.

These reduction rules may be built into decomposition trees that are similar to those presented for SP graphs. However, these trees are no longer binary and will have two types of nodes: separator sets and reduction rules. A reduction rule contains the vertex it eliminated, v_i, and is also associated with a separator set node which contains $adj(v_i)$. We can represent reduction rules graphically as shown in Figure 29.

After application of a reduction rule, the graph has one fewer vertex, and a decomposition tree fragment has been constructed. These fragments are joined

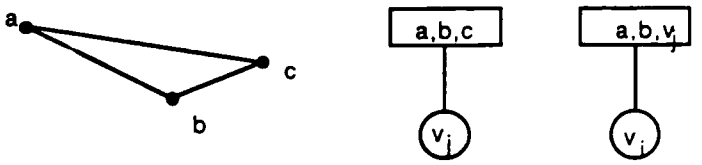


Figure 30: Elimination of v_j and the Resulting Decomposition Tree Fragments

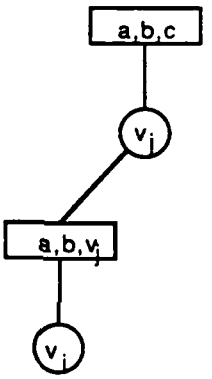


Figure 31: The Resulting Decomposition Tree

into a tree structure whenever possible. Consider elimination of v_j as shown in Figure 30. There are now two decomposition tree fragments. Through application of Theorem 6,it is known that since $v_j \in adj(v_i)$, $adj(v_j)$ separates v_i from the graph as well. This relationship can be reflected naturally in the joining of tree fragments as shown in Figure 31.

An algorithm for the construction of a decomposition tree is presented in Figure 32. Initially, there is a graph G and an empty set of tree fragments, F. Application of a reduction rule to a vertex v results in a new tree fragment with a separator set node S, and reduction rule node R. Any existing fragments in F whose root (which is a separator set node) contains v is a candidate for joining as described above. At the termination of the algorithm, there is one tree fragment in F, which

is a decomposition tree for the entire graph.

This algorithm poses an interesting problem. The order of selection of the vertices determines the separator sets found as well as their size. Since it is desired to find separator sets of minimum size, the problem is identical to that of minimizing $d(v)$ in Gaussian elimination. Once again, the minimum degree algorithm is a good heuristic that minimizes $d(v)$ for all v in the types of graph with which this report is most concerned: GSP graphs and other graphs that may be reduced by rules of small order.

5.3 A Separator Theorem for a Bipartite Tree

A theorem presented for an unusual tree will later be useful for finding separators for arbitrary graphs. This tree has two types of nodes, **red** and **black**. A **red** node, r, has internal weight, $i(r) = 1$, and a **black** node, b, has internal weight, $i(b) = 0$. The total weight of a node n (of either type) is the sum of the internal weights of all descendants of n.

$$w(n) = \sum_{p \in descendants(n)} i(p)$$

Each of these nodes may have any number of children, which must be of the opposite type; thus, it is a bipartite tree. The root of the tree is a **black** node and leaves of the tree are **red** nodes. Figure 33 shows a sample **red-black** tree with appropriate weights labeled. **Red** nodes are represented by circles, and **black** nodes by squares.

The weight of the root node $w(root) = W$ indicates how many **red** nodes there are in the tree. If the number of children of all **red** nodes in the tree is bounded by some constant, k, then we can show that there is a **black** node whose removal from the tree causes the tree to split into parts, none of which will have weight greater than $\frac{k}{k+1}W + \frac{1}{k+1}$.

Theorem 7 *A* **Red-Black** *tree of total weight W in which no red node has more than k children has a* **black** *node b that separates the tree into parts, each of which has weight less than or equal to $\frac{k}{k+1}W + \frac{1}{k+1}$.*

```
{Given G = ⟨V, E⟩ }

type SeparatorSet = record
      set, sons
endrecord;

type ReductionRule = record
      vertex, order, sons
endrecord;

ReductionRule R;
SeparatorSet S;

F ← ∅
for i = 1 to n do
      V ← V − v_{α_i};
      E ← E ∪ D(v_{α_i});
      new R; S ← null;
      R.vertex ← v_{α_i}; R.order ← d(v_{α_i}); R.sons ← ∅;
      {See if a separator set with these elements has already been found}
      for each SeparatorSet t ∈ F do
            if adj(v_{α_i}) ⊆ t.set and S ≠ null then t.sons ← t.sons ∪ { R }; S ← t fi
      od
      if S = null then
            new S; S.set ← adj(v_{α_i}); S.sons ← { R }
      fi

      { Try to find sets to add as descendants }
      for each SeparatorSet t ∈ F do
            if v_{α_i} ∈ t.set then
                  R.sons ← R.sons ∪ {t};
                  F ← F−t;
            fi
      od
      F ← F∪ S;
od
```

Figure 32: An Algorithm to Produce a Decomposition Tree for an Arbitrary Graph

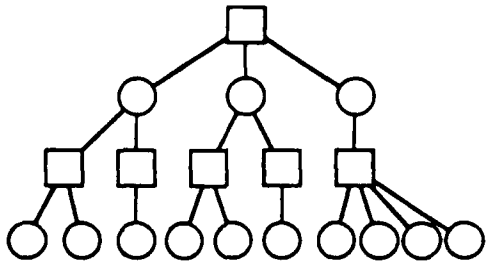


Figure 33: A Red-Black Tree

PROOF: Black nodes are examined starting at the root of the tree and descending through the tree. It is obvious that some black node, b_1, can be found whose weight meets the $\frac{k}{k+1}W + \frac{k}{k+1}$ criterion, but it must also be shown that the total weight of all nodes not descendants of the current black node, $W - w(b_1)$, does not exceed the criterion.

INITIALIZATION: b_1 is the root of the tree.

INVARIANT: $W - w(b_1) \leq \frac{k}{k+1}W + \frac{1}{k+1}$. This is true for the initial case since $W = w(b_1)$.

GENERAL STEP: Find the child of b_1 with the greatest weight. Call this node r. If $w(r) \leq \frac{k}{k+1}W + \frac{1}{k+1}$, then b_1 is the node that satisfies the required criterion.

Otherwise, find the largest child of r and call it b_2. If $w(b_2) > \frac{k}{k+1}W + \frac{1}{k+1}$, then repeat the general step with $b_1 \leftarrow b_2$. The invariant still holds.

Otherwise, it is claimed that b_2 is the black node that separates the tree as required. It is known that $w(b_2)$ meets the criterion. It remains to be shown that the weight of all nodes that are not descendants of b_2, $W - w(b_2)$, does not exceed the criterion. Since a red node may have at most k children, it is known that

$$w(b_2) \times k + 1 \geq w(r) > \frac{k}{k+1}W + \frac{1}{k+1}.$$

Therefore,

$$w(b_2) > \frac{1}{k+1}W + \frac{1}{k(k+1)} - \frac{1}{k}$$

and,

$$w(b_2) > \frac{1}{k+1}W - \frac{1}{k+1}.$$

Finally,

$$W - w(b_2) \leq W - (\frac{1}{k+1}W - \frac{1}{k+1})$$

$$W - w(b_2) \leq \frac{k}{k+1}W + \frac{1}{k+1}$$

□

An actual implementation of the algorithm as described appears in Figure 34. This algorithm simply implements the tests described in the proof. It will be used later in this chapter as part of another algorithm.

5.4 Using a Decomposition Tree to find Separators

In the decomposition tree proposed, each separator set node is a valid separator of the graph for some value of α. This section will examine how a good separator for the graph may be selected, and what values of α can be expected depending on the reduction rules applied.

The decomposition tree proposed in this chapter is in fact a red-black tree. Separator set nodes correspond to black nodes, reduction rule nodes to red nodes. The weights proposed for red-black trees calculate the number of reduction rules under any given node in the decomposition tree, and hence, give the number of vertices of the graph separated by a given separator set. Removal of a black node in the tree corresponds to the removal of the separator set from the graph. The number of nodes in each remaining part of the graph is the total weight of the corresponding part of the decomposition tree. Theorem 7 states that this can be bounded for some value of k. For a decomposition tree, k is the maximum number of children of any reduction rule in the tree. It will be shown that k is dependent on the reduction rules applied.

```
function RedBlack(tree T, int k, int W) returns tree;
    if red(T) then
        max ← ∅;
        for p ∈ children(T) do
            if w(p) > w(max) then max ← p
        od
        RedBlack(max, k, W);
    else
        {Are at a black node }
        if w(T) ≤ k/(k+1) W + 1/(k+1) then
            return T;
        else
            max ← ∅;
            for p ∈ children(T) do
                if w(p) > w(max) then max ← p
            od
            if w(p) ≤ k/(k+1) W then
                return T;
            else
                RedBlack(max, k, W);
            fi
        fi
    fi
end RedBlack;
```

Figure 34: Algorithm to Find a Black Separator Node

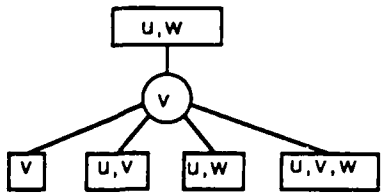


Figure 35: A Reduction Rule of Order Two

The number of children of any reduction rule is directly related to the *order* of the reduction rule applied. (R_0 is called order 0, R_1 order 1, R_2 order 2, etc.) Consider a reduction rule of order two as shown in Figure 35. The parent node of this rule must be a separator set containing two vertices. The only possible separator sets that could be children of the reduction rule are those that include the vertex, v, and some subset of the parent separator set, $\{u, w\}$. The four possible subsets are: $\{v\}, \{u, v\}, \{v, w\}$, and $\{u, v, w\}$. For any reduction rule of order k, there may be, at most, 2^k separator sets as children.

5.4.1 Bounding the Number of Children

The previous section stated that a reduction rule of order k could have, at most, 2^k children. This is true only in the worst case. Consider a graph in which all reduction rules are of order two (an SP graph for example). While each reduction rule has the potential for having four children, each will have at most two, since each of its children must be sets containing two vertices. In general, if a graph can be described entirely by reduction rules of order k, then each reduction rule will have, at most, k children.

For decomposition trees in which there are reduction rules of varying orders, the number of children of a reduction rule may be limited by merging some of its child separator sets together. This may increase the size of the separator sets, but decreases the number of children of the reduction rule.

As stated earlier, a reduction rule may have as children the separator sets whose elements are the eliminated vertex and some subset of the parent separator set. Of

the 2^k separator sets which may be children of the reduction rule of order k, k of them are "maximal" subsets of k elements in the sense that all other separator sets of k or fewer elements are subsets of one of these sets. If there is no reduction rule in the tree of order $k+1$, then it is known that these k separator sets will be the only children of the reduction rule.

The algorithm of Figure 36 is similar to the algorithm of Figure 32 but ensures the merging of separator sets by creating the k maximal subsets of a reduction rule as its children. A procedure called **MakeSubsets** is used which constructs the set of maximal separator sets. When an existing separator set is found which is a subset of any of these maximal subsets, it is simply merged into the set, adding its children to the list of children of the newly created maximal subset. In the event the existing separator set is not a subset of any of the newly created maximal subsets (in which case it must be the one possible subset of $k+1$ elements), it is simply added as a child of the reduction rule. By applying this process, decomposition trees may be created which obey the following theorem.

Theorem 8 *In the decomposition tree for a graph described entirely by reduction rules of order k or less, each reduction rule will have, at most, k children.*

This theorem and combined with Theorem 7 leads to the following:

Lemma 1 *A graph described entirely by reduction rules of order k or less has a k-separator with* $\alpha = \frac{k}{k+1} + \frac{1}{n(k+1)}$ *and* $\beta = 1$.

5.5 A Nested Dissection Ordering Algorithm

The decomposition tree can be used to produce an elimination ordering in much the same manner as the algorithm proposed for decomposition trees of SP graphs. The algorithm of Figure 37 takes as arguments a decomposition tree T, and a separator criterion k. Auxiliary procedures are defined as follows:

RedBlack: Finds the separator set node in the decomposition tree that meets the

```
{Given G = ⟨V, E⟩ }

type SeparatorSet = record
     set, sons
endrecord;

type ReductionRule = record
     vertex, order, sons
endrecord;

ReductionRule R; SeparatorSet S;

F ← ∅
for i = 1 to n do
     V ← V − v_{α_i};
     E ← E ∪ D(v_{α_i});
     new R; S ← null;
     R.vertex ← v_{α_i}; R.order ← d(v_{α_i}); R.sons ← MakeSubsets(v_{α_i}, adj(v_{α_i}));
     {See if a separator set with these elements has already been found}
     for each SeparatorSet t ∈ F do
          if adj(v_{α_i}) ⊆ t.set and S ≠ null then t.sons ← t.sons ∪ { R }; S ← t fi
     od
     if S = null then
          new S; S.set ← adj(v_{α_i}); S.sons ← { R }
     fi

     { Try to find sets to add as descendants }
     for each SeparatorSet t ∈ F do
          if v_{α_i} ∈ t.set then
               { See if there is a superset present }
               bool found = FALSE;
               for each SeparatorSet s ∈ R.sons do
                    if t.set ⊆ s.set and not found then
                         s.sons ← s.sons ∪ t.sons; found = TRUE;
                    fi
               od
               if not found then R.sons ← R.sons ∪ {t};fi
               F ← F−t;
          fi
     od
     F ← F∪ S;
od
```

Figure 36: Modified Algorithm to Produce Decomposition Tree for an Arbitrary Graph

```
procedure GeneralizedOrdering( tree  T, int  k);
      if T is a leaf reduction rule then
            Number(T.vertex);
            return;
      fi
      p ← RedBlack(T, k, w(T));
      for each v ∈ T.separator do
            Number(v);
      od
      S ← p.sons;
      {Remove this separator set node from the tree}
      (p.parent).sons = (p.parent).sons − p;
      RelabelParents(p);
      GeneralizedOrdering(T);
      for each p ∈ S do
            GeneralizedOrdering(p);
      od
end GeneralizedOrdering;
```

Figure 37: Elimination Ordering Algorithm for Generalized Decomposition Trees

given k value and returns this node.

Number: Receives a vertex as an argument and assigns it an elimination number. This function begins numbering at n and number vertices in a descending order. The function "remembers" those vertices that have already been numbered, and does not re-number them.

RelabelParents: The tree is assumed to have weights initially calculated for each node. Removal of a node requires relabeling of the weights of only the nodes above it in the tree. This procedure follows parent links to perform this renumbering.

5.6 A Bound on Complexity

The total number of operations in a DAG produced by this method is a function of the largest reduction rule applied. Consider a graph which is described by reduction rules, all of which are of order k or less. Each separator set will have k or fewer vertices, as will the subgraphs defined by the set of vertices not belonging to any separator set. The elimination degree of each of the vertices belonging to these subsets can be no more than $d(v) < 3k$ for the following reason. Each of the vertices can be adjacent to two separator sets which have no more than k vertices, and may also be adjacent to vertices in its own subgraph of which there are fewer than k vertices. Thus, each of these vertices will have $d(v) \leq 3k$.

Similarly, at the time of their elimination, each of the vertices in the separator sets may be adjacent to two separator sets (of k vertices), and to the other vertices of its own separator set. Thus, each of these vertices has $d(v) \leq 3k$. These facts, coupled with Equation 1, gives the total complexity of performing Gaussian elimination by this method as no more than

$$(9k^2 + 3k)n,$$

which is

$$O(k^2 n).$$

Thus, k may be viewed as measuring the sparsity of a graph. Small values of k occur for graphs which are sparse, while the largest value k may take on, n, occurs for a complete graph for which the complexity of performing Gaussian elimination is known to be $O(n^3)$. The expression above correctly captures this fact.

5.7 A Bound on Depth

The DAG generated by Gaussian elimination has depth dependent on the manner in which vertices are eliminated. The generalized nested dissection algorithm presented in this chapter can be analyzed for bounds on the depth of DAGs generated.

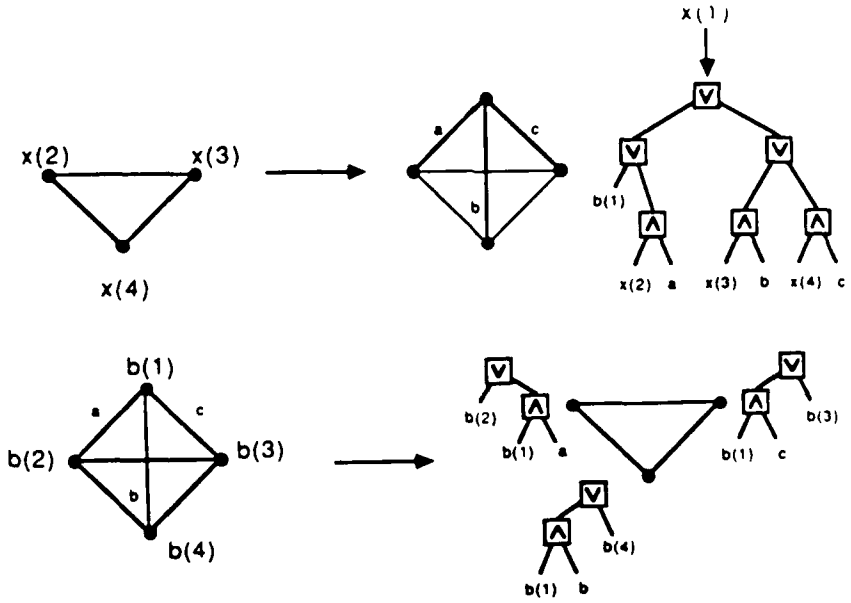


Figure 38: The Depth of DAGs for a Clique

The elimination of the vertices of a clique results in a total amount of depth added to the DAG as described by the following.

$$d_{clique}(k) = d_{clique}(k-1) + 3 + \lceil \lg k \rceil$$

$$d_{clique}(2) = 4$$

The calculation for a clique of two vertices is trivial. The other part of the recurrence will be explained with the aid of Figure 38, which illustrates a clique of four vertices. Elimination of vertex 1 results in the DAGs under construction for vertices 2,3 and 4 growing deeper by two operators. The second half of the figure shows what happens during backsubstitution. The edges a, b and c each contribute a depth of 1 to the DAG of vertex 1 with an $\vee$ operator, while the $\wedge$ operators may be balanced with total depth $\lceil \lg 4 \rceil$. Thus, the amount added in the recurrence is $(2 + 1 + \lceil \lg n \rceil)$.

A separator divides the graph into two or more subgraphs. Prior to its elimination, all of the vertices of the two subgraphs have been eliminated except for one in each subgraph, as depicted in Figure 39. Elimination of these two vertices external to the separator results in a clique formed among the vertices of the separator. The depth of DAGs for a clique has already been explored; the addition of two external vertices must be considered next. If the separator is of size k, then the separator with its two external vertices almost form a clique of $k + 2$ vertices, except that there is no edge between the two external vertices. This recurrence makes use of

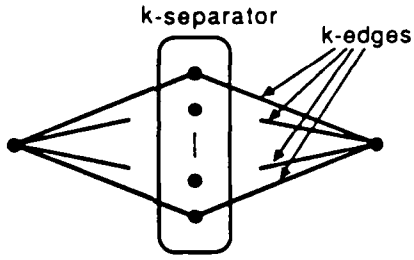


Figure 39: A Separation Set and Two External Vertices

the formula for describing the depth of a clique.

$$d_{sep}(k) = d_{clique}(k) + 4 + \lceil \lg(k+1) \rceil$$

The value of $d_{sep}(k)$ has been calculated for a few values of k as shown in the table below. The function grows very slowly; its asymptotic growth is $O(k \lg k)$.

k	$d_{sep}(k)$
1	5
2	10
3	15
4	21
5	28
6	35
7	42
8	50

From the decomposition tree for a given graph, the size of the maximum reduction rule applied, k, can be found. This value also bounds the best value of α that can be achieved using the separators shown in the decomposition tree. Each of the subgraphs defined by the separator will be eliminated to a single vertex in a similar manner, with a constant amount of depth added to the DAG. The size of the largest subgraph is guaranteed to be roughly $\frac{k}{k+1}$ of the total graph.

The following recurrence describes the manner in which depth grows for a given number of vertices in a graph, n, and the maximum sized separator, k. $D_k(n)$ is a function that calculates the upper bound on the depth of DAG generated for a given number of vertices with a maximum reduction rule of order k.

$$D_k(n) = D_k(\frac{k}{k+1}n) + d_{sep}(k)$$

The solution to this recurrence is given by

$$D_k(n) = \sum_{1 \le i \le \log_{\frac{k+1}{k}} n} d_{sep}(k)$$

which is

$$O(\lceil k \lg k \rceil \log_{\frac{k+1}{k}} n)$$

Thus, sublinear growth of the depth of DAG can be guaranteed with this method. However, as k gets large, the coefficients increase and the logarithmic growth of depth degrades.

5.8 SP and GSP Graphs Revisited

The separator theorem developed in this chapter can be seen to be a generalization of the separator theorem for series-parallel graphs. SP graphs can be completely described by rules of order k or less with $k = 2$. Therefore, for SP graphs the generalized decomposition tree finds 2-separators with $\alpha = \frac{2}{3}$.

General Series-Parallel graphs were similar to SP graph except that they included acyclic branches. This class of graph is important in that they may be used to describe most MOS circuits that arise. GSP graphs may be described by reduction rules of order 1 and 2. Therefore, 1- and 2-separators for GSP graphs with $\alpha = \frac{2}{3}$ may be found, and the DAGs representing the solution of such a network will have depth that grows as $O(\lg n)$.

5.9 Experimental Results

The generalized nested dissection algorithm described in this chapter has been implemented and tested for a variety of graphs that occur in MOS networks. Channel graphs for some of the networks are shown in Appendix B. The DAGs generated were assumed to represent binary Boolean operators and were analyzed for their minimum depth. DAGs may be rearranged to decrease their depth through a very simple procedure. Often, the elimination process results in a long chain of $\vee$ operators that may be rebalanced into a shorter tree. This was accounted for in the program that analyzed the minimum depth of a DAG.

For all comparisons, the results of a minimum degree ordering algorithm along with the results of the generalized nested dissection algorithm have been shown. The minimum degree algorithm has been observed to minimize the total number of operations for most of the graphs presented, while the generalized nested dissection algorithm tends to minimize depth. In most cases, there is a tradeoff between these two: a decrease in depth results in higher total complexity.

This first group of graphs contains linear chains of the number of vertices indicated in Table 2. While chains of the lengths presented do not occur often in real networks, they dramatically illustrate the tradeoff between depth and complexity.

DAGs resulting from the minimum degree algorithm are essentially a straight line of operators in which only one operation may be performed at a time. However, at the increased expense of more total operations, the DAG produced by the generalized nested dissection algorithm has significantly less depth. For the longest chain of 100 nodes, the DAGs differ in depth by almost a factor of 10, indicating that a machine capable of parallel evaluation could evaluate the response of a network described by this graph ten times faster than a general purpose computer.

Shift networks present a difficult problem for Gaussian elimination. No algorithms that result in a low value of total operations have been found. Table 3 shows the results obtained for three different sixteen bit shifters that rotate data one of three possible bit positions. Usually, the generalized nested dissection algorithm resulted in lower depth with nearly the same total number of operations. These

Chain Length	Minimum Degree Algorithm		Generalized Nested Dissection	
(n)	depth	complexity	depth	complexity
10	36	36	17	60
20	76	76	25	164
30	116	116	27	284
40	156	156	33	388
50	196	196	35	308
60	236	236	35	628
70	276	276	41	732
80	316	316	41	852
90	356	356	41	972
100	396	396	43	1092

Table 2: Results for Linear Chains of Varying Length

Shifter	Minimum Degree Algorithm		Generalized Nested Dissection	
	depth	complexity	depth	complexity
shft16-014	112	1632	100	1632
shft16-012	126	904	92	924
shft16-013	130	1364	109	1364

Table 3: Results for 16-bit shifters

results are encouraging. While the savings in depth were not large, there was no extra cost in total operations for two of the shifters and very little for the third.

Logical shifters do not exhibit the same level of complexity for solution that rotational shifters do. The results are shown in Table 4 and are very interesting with respect to the tradeoff between depth and complexity. Two of the networks could be solved with a significant savings in depth with a corresponding increase in complexity. The other two were solved with negligible savings in depth, but with no increase in total operations.

A number of RAM cells are shown in Table 5. Each of these networks can be described by reduction rules of order 1. Parity ladders of varying numbers of nodes

Shifter	Minimum Degree Algorithm		Generalized Nested Dissection	
	depth	complexity	depth	complexity
lshft16-01	128	128	29	304
lshft16-012	174	372	64	748
lshft16-014	97	832	92	832
lshft16-018	77	616	74	616

Table 4: Results for 16-bit logical shifters

RAM	Minimum Degree Algorithm		Generalized Nested Dissection	
	depth	complexity	depth	complexity
ram4	28	32	13	40
ram16	39	100	20	132
ram32	41	164	21	196
ram64	50	340	23	436
ram256	61	1220	31	1476

Table 5: Results for RAMs of Varying Sizes

are shown in Table 6. Each of these may be described by reduction rules of order 3, and hence, have 3-separators. The results of performing Gaussian elimination on both of these types of networks shows a logarithmic growth of depth. These results indicate that the algorithm does, in fact, perform well for graphs that can be described by small reduction rules.

A few miscellanous networks are shown in Table 7. *Seradd.a* and *seradd.b* represent two separate subnetworks in a serial adder circuit, and *par4* is a parity generator for four bits. These results once again illustrate the tradeoff between depth and complexity, indicating that the generalized nested dissection algorithm degrades to a level of performance comparable to that of the minimum degree algorithm for graphs without good separators.

Parity Ladder	Minimum Degree Algorithm		Generalized Nested Dissection	
	depth	complexity	depth	complexity
10	37	62	39	76
20	92	152	85	220
30	147	242	97	376
40	202	332	113	606
50	257	422	150	940
60	312	512	155	1160

Table 6: Results for Parity Ladders of Varying Number of Vertices

Network Name	Minimum Degree Algorithm		Generalized Nested Dissection	
	depth	complexity	depth	complexity
seradd.a	23	64	23	64
seradd.b	17	24	10	24
par4	25	88	24	88

Table 7: A Few Random Networks

Chapter 6

Discussion

This report has analyzed the expressions generated by Gaussian elimination with a metric not widely discussed. While the depth of an expression has not been an issue in the past, the introduction of highly parallel hardware makes it an important consideration. It was shown that depth can be minimized by the proper selection of an elimination ordering through nested dissection and that the growth of depth can be bounded as a function of characteristics of the graph.

Two separate topics have been examined in this report. One is a method for solving systems of Boolean equations, and the other is a method of finding separators in arbitrary graphs. It is curious that the general framework of Gaussian elimination is common to the solution of both problems. The use of separators to produce an elimination ordering for Gaussian elimination results in a strange double use of the general Gaussian elimination algorithm: once to find separators and then again to actually solve the system. In fact, the actual code was written as two identical Gaussian elimination shells, with operators replaced as necessary.

The method presented for computing an elimination ordering is especially suited to those graphs that can be completely decomposed using reduction rules of small order. GSP graphs were shown to be able to be eliminated with all vertices having elimination degree of two or less, resulting in efficient solution of systems described by GSP graphs with small total depth. This is fortunate, since the original application was the simulation of MOS networks, most of which can be described by GSP

graphs.

While the main intent of this work was to find a way of optimizing the DAGs describing MOS networks for depth, the method is applicable to any graph. The order of reduction rules applied, k, was shown to determine the depth of the resulting DAG. Experimental evidence also showed that there seemed to be a tradeoff between depth and total complexity. For graphs that did not have good separators by the method, the resulting DAG was similar in depth and complexity to that produced by the minimum degree elimination algorithm, which is considered to be good for many classes of graphs. This is a desirable characteristic, indicating that the algorithm proposed degrades gracefully, rather than simply not working at all.

6.1 Future Considerations

This work has raised a number of interesting questions. The basic algorithm used in the production of a decomposition tree is the minimum degree algorithm; other methods should be examined. In particular, a nested dissection algorithm should be applied at this stage to produce an elimination ordering for finding the decomposition tree. This tree would then be used to produce an elimination ordering to solve the system.

Finally, there may be other methods of explicitly minimizing the depth of expressions. A method based on a greedy depth algorithm was explored and looked promising, but the results were inconclusive. While separators divide the graph, they do not guarantee that the subgraphs induced have nearly the same "span", or length of longest path in the graph. Depth is clearly related to how far (in edges) information travels in a graph. Explicit minimization of this factor could lead to DAGs of even smaller depth.

Appendix A

Graph Theoretical Definitions

A short introduction to some basic terms from graph theory will be given here. Most are standard (see [17]).

A graph $G = \langle V, E \rangle$ consists of a finite set of $n = |V|$ *vertices* and a finite set of $m = |E|$ *edges*. Edges are pairs of vertices with $E \subseteq \{(v, w) | v, w \in V, v \neq w\}$. If (v, w) is an edge, vertices v and w are *adjacent* and edge (v, w) is incident to v and w, which are its endpoints. The number of edges incident on a vertex is its *degree*. The set of vertices adjacent to v are denoted $adj(v)$ and are sometimes also called the *neighbors* of v.

If the edges are unordered pairs then the graph is *undirected*, otherwise it is a *directed* graph. An edge (v, w) in a directed graph has a *tail* v and a *head* w. A directed graph is called a *digraph*. A vertex v of a digraph has an associated *indegree* and *outdegree*. The number of edges whose head is v is the vertex's indegree and the number of edges whose tail is v is its outdegree. A vertex whose indegree is 0 is called a *source* and a vertex whose outdegree is 0 is called a *sink*.

If the set of edges is a multiset, that is, if multiple edges between the same two vertices are allowed, then the graph is called a *multigraph*. A directed multigraph is called a *multidigraph*.

Graph $G' = \langle V', E' \rangle$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. If $W \subseteq V$, the *induced subgraph* $G(W) = \langle W, E(W) \rangle$ where

$$E(W) = \{(v, w) \in E | v, w \in W\}.$$

A complete graph is a graph in which each pair of distinct vertices is joined by an edge. A *clique* of a graph G is a subset S of V such that $G(S)$ is complete.

A *path of length k* between vertices v and w is a sequence of vertices $v = v_0, v_1, \cdots, v_k = w$ such that $\{v_{i-1}, v_i\}$ is an edge for $1 \leq i \leq k$ and all the vertices $v_1, \cdots, v_k$ are distinct. If $v = w$ the path is a *cycle*. If every pair of vertices in G is joined by a path, then G is *connected*. A *chord* is an edge that joins two vertices in a cycle that are not adjacent. A *chordal* graph is one in which every cycle of at least four vertices has a chord

If vertices of a graph can be partitioned into two sets V_1 and V_2 such that every edge has one endpoint in V_1 and the other in V_2, the the graph is a *bipartite* graph. Similarly, if the nodes of a tree can be partitioned into two sets such that every edge has one endpoint in each of the sets, then the tree is a bipartite tree.

Appendix B

Various Channel Graphs

Figure 40: A Linear Chain Graph of Ten Vertices

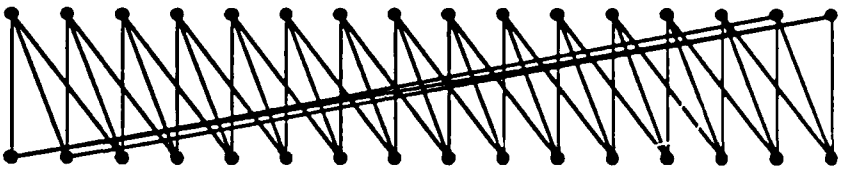


Figure 41: A Sixteen-Bit Shifter

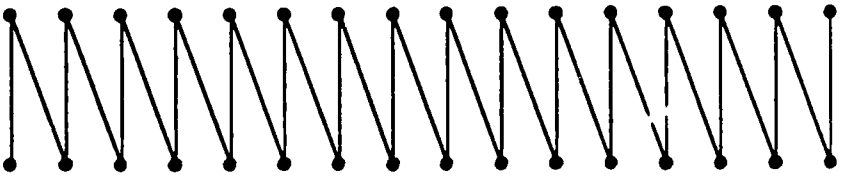


Figure 42: A Sixteen-Bit Logical Shifter

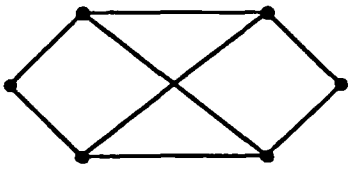


Figure 43: Seradd.a

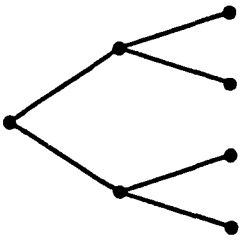


Figure 44: Seradd.b

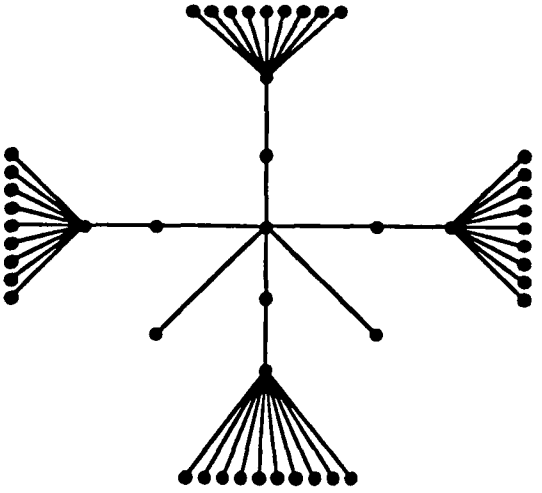


Figure 45: Channel Graph for a Sixteen-Bit RAM Cell

Bibliography

[1] *ZyCad LE-001 and LE-002 Product Description.* ZyCad Corp., 1982.

[2] S. K. Abdali and B. D. Saunders. Transitive Closure and Related Semiring Properties via Eliminants. *Theoretical Computer Science*, 40:257–274, 1985.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[5] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* The Macmillan Press Ltd., 1982.

[6] R. E. Bryant. Conversation, 1986.

[7] R. E. Bryant. *Papers about a Symbolic Analyser for MOS Circuits.* Computer Science CMU-CS-86-114, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, March 1986.

[8] W. J. Dally and R. E. Bryant. A Hardware Architecture for Switch-Level Simulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, CAD-4(3):239–249, July 1985.

[9] M. M. Denneau. The Yorktown Simulation Engine. In *19th Design Automation Conference*, pages 55–59, ACM, 1982.

[10] H. N. Djidjev. Separator Theorems for Planar Graphs. *Doklady Bolgarskoi Akademii Nauk*, 34(2):163–164, 1981.

[11] R. J. Duffin. Topology of Series-Parallel Networks. *Journal of Math. Anal. and Appl.*, 10:303–318, 1965.

[12] E. H. Frank. Switch-Level Simulation of VLSI Using a Special-Purpose, Data-Driven Computer. In *22nd Design Automation Conference*, pages 735–738, ACM, 1985.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, 1979.

[14] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.

[15] A. George and J. W. H. Liu. An Automatic Nested Dissection Algorithm for Irregular Finite Element Problems. *SIAM Journal on Numerical Analysis*, 15:1053–1069, 1978.

[16] J. R. Gilbert. *Graph Separator Theorems and Sparse Gaussian Elimination.* PhD thesis, Stanford University, December 1980.

[17] F. Harary. *Graph Theory.* Addison-Wesley, 1969.

[18] J. Hopcroft and R. E. Tarjan. Dividing a Graph into Tri-connected Components. *SIAM J. Computing*, 2(3):135–158, Sept 1973.

[19] J. A. G. Jess. Some New Results on Decomposition and Pivoting of Large Sparse Systems of Linear Equations. *IEEE Trans. on Circuits and Systems*, 23(12):729–738, December 1976.

[20] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized Nested Dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, April 1979.

[21] R. J. Lipton and R. E. Tarjan. Applications of a Planar Separator Theorem. *SIAM Journal on Computing*, 9(3):615–627, August 1980.

[22] L. W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits.* PhD thesis, Univ. of California, Berkeley, May 1975.

[23] V. Pan and J. Reif. *Efficient Parallel Solution of Linear Systems.* Technical Report TR-02-85, MIT, 1985.

[24] V. Pan and J. Reif. *Extension of the Parallel Nested Dissection Algorithm to the Path Algebra Problems.* Technical Report TR-15-85, MIT, 1985.

[25] S. Parter. The Use of Linear Graphs in Gaussian Elimination. *SIAM Review*, 3(2):119–130, 1961.

[26] D. J. Rose. A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations. In R. C. Read, editor, *Graph Theory and Computing*, pages 184–218, Academic Press, 11 Fifth Ave, New York, NY 10003, 1972.

[27] D. J. Rose and R. E. Tarjan. Algorithmic Aspects of Vertex Elimination on Directed Graphs. *sicomp*, 34(1):176–197, January 1978.

[28] C. E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Trans. of the AIEE*, 57:713–723, 1938.

[29] R. E. Tarjan. Fast Algorithms for Solving Path Problems. *J. ACM*, 28(3):595–614, July 1981.

[30] J. Valdes, R. E. Tarjan, and E. L. Lawler. The Recognition of Series Parallel Digraphs. *11th Annual ACM Symposium on Theory of Computing*, 6:1–12, May 1979.

[31] J. A. Valdes. *Parsing Flowcharts and Series-Parallel Graphs.* PhD thesis, Stanford University, 1978.

[32] M. Yannakakis. Computing the Minimum Fill-in is NP-Complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.

END

FILMED

MARCH, 1988

DTIC